Microprocesseurs

et Microcontrôleurs

Représentation de l'information en numérique

Jlassi Khaled

Université Virtuelle de Tunis

2008

Ce module « Microprocesseurs et Microcontrôleurs » s'intéresse aux systèmes microprogrammés à base de microprocesseurs et/ou de microcontrôleurs.

Les sujets choisis, le niveau auquel ils sont abordés, ont été sélectionnés de manière à coïncider avec les cours enseignés au niveau de certaines filières d'ingénieurs (Génie Électrique, Télécommunication, Informatique et certaines maîtrises technique).

Concevoir une interface matérielle, pour un microprocesseur ou une application spécialisée, et arriver à un résultat raisonnablement optimisé, nécessite des connaissances assez vastes, et il faut sans cesse les compléter. La maîtrise des méthodes traditionnelles des systèmes logiques est insuffisante pour la conception d'un système à base de microprocesseur ou microcontrôleur. La pratique de plusieurs réalisations est nécessaire pour acquérir une connaissance réelle. Toutefois ce cours a pour ambition de montrer à de jeunes étudiants et à des ingénieurs désireux de se mettre à jour et recycler leurs connaissances sur un sujet en perpétuelle évolution les facettes principales de la conception de systèmes à microprocesseur. Les sujets abordés ici sont similaires à ceux qui sont le plus fréquemment traités dans certains ouvrages de références et certains sites Internet. Nous avons volontairement découpé ce cours en cinq parties indépendantes mais complémentaires. L'étudiant ou l'étudiante devrait avoir une connaissance de base en systèmes logiques et informatique. Toutefois Dans la première partie nous avons fait un rappel sur les systèmes numériques. C'est un pré requis nécessaire pour le reste du cours

Ce « Guide d'étude » a pour objectif de vous préparer à suivre le cours. Il définit en quelque sorte un mode d'emploi, non seulement pour le matériel didactique du cours, mais aussi pour le cheminement que vous devez adopter et les différentes exigences auxquelles vous devez répondre.

Bonne lecture et bon cours!

Le but de ce module est de se familiariser avec les principes de base des circuits numériques. Plus spécifiquement ce cours a pour ambition de montrer, à de jeunes étudiants et à des ingénieurs désireux de se mettre à jour et recycler leurs connaissances sur un sujet en perpétuelle évolution, les facettes principales des systèmes microprogrammés.

Au terme de ce cours les étudiants seront en mesure :

- **d'expliquer les différente représentations des données numérique et alphanumériques.**
- **d'analyser et pourquoi pas de concevoir des systèmes minimales a base de microprocesseurs; quoique la pratique de plusieurs réalisations est nécessaire pour acquérir une connaissance réelle de conception de systèmes à base de microprocesseurs.**
- **de proposer certaines solutions relatives à des applications simples de commande et de contrôle impliquant un microprocesseur ou un microcontrôleur.**
- **de donner un avis professionnel sur les différentes familles de microcontrôleurs et leurs champ d'application.**
- **de définir des critères de choix d'un microcontrôleur pour une application donnée. En effet il existe plusieurs famille de microcontrôleurs dont les plus connues sont : Atmel AT91 , Atmel AVR , le C167 de Siemens/Infineon, Hitachi H8, Intel 8051,Motorola 68HC11, PIC de Microchip, ST6 de STMicroelectronics, ADuC d' Analog Devices, PICBASIC de Comfile Technology . Il est bien évident que, dans le cadre de ce cours dont le nombre de pages doit forcément rester limité, il ne va pas être possible de donner toutes les informations, matérielles et logicielles, relatives à tous ces microcontrôleurs. Le manuel technique de chacun d'entre eux comporte en effet plusieurs dizaines de pages, voir parfois une centaine. Nous allons nous intéresser dans le cadre de ce cours à la famille Intel C51. Après une brève présentation de cette famille nous avons orienté notre étude vers le microcontrôleur 80C51. C'est un microcontrôleur 8 bits avec un jeu d'instructions réduit (101 instructions). Ses instructions sont organisées autour d'un accumulateur et de registres (quatre banques de huit registres). L'unité centrale du 80C51 incorpore un processeur booléen qui accroît considérablement la vitesse de traitement des instructions de manipulation de bits.**

Nous avons consacre la dernière partie de ce cours à la programmation des microcontrôleurs et notamment la famille C51. Pour cela nous avons pris le cas du

microcontrôleur 80C552 de PHILPS. Ce microcontrôleur incorpore une liaison série RS232S et un bus I2C (réseau local) et 8 entrées analogiques. L'environnement de programmation que nous avons choisi pour illustrer nos exemples et l'environnement Keil. Cet environnement extrêmement puissant présente une plateforme logicielle complète pour la programmation des microcontrôleurs. L'environnement Keil intègre un débogueur avec µVision. Il peut être utilisé en mode simulateur ou moniteur temps réel chargé sur cible. Le dialogue se faisant à travers une liaison série ou USB suivant les architectures.

Le déboguage est facilité grâce à une intégration complète des microcontrôleurs, on a ainsi une visibilité sur l'état de tous les périphériques internes en mode simulation. Keil fournit tous les fichiers de démarrage (Startup, header..) pour chacune des architectures ainsi que de nombreux projets d'exemples utilisant le CAN, l'I2C, l'Ethernet et les périphériques internes. Le moniteur temps réel est pré-configuré pour la plupart des cartes d'évaluation du marché.

Grâce à son Editeur de liens étendue (LX51), Keil supporte aujourd'hui tous les dérivés 8051 ayant un adressage supérieur à 64K. Un RTOS est aussi fourni en version allégée et peut être complété optionnellement (RTX 51, RTX 166 ou ARTX sur ARM, suivant l'architecture). Il supporte les dernières architectures XC166 d'Infineon et Super 10 de ST Microelectronics, incluant leurs fonctionnalités DSP.

I. Présentation du binaire.

Vers la fin des années 30, Claude Shannon démontra qu'à l'aide de "contacteurs" (interrupteurs) fermés pour "vrai" et ouverts pour "faux" il était possible d'effectuer des opérations logiques en associant le nombre " 1 " pour "vrai" et "0" pour "faux". Ce codage de l'information est nommé base binaire. C'est avec ce codage que fonctionnent les ordinateurs. Il consiste à utiliser deux états (représentés par les chiffres 0 et 1) pour coder les informations. L'homme travaille quant à lui avec 10 chiffres (0,1,2,3,4,5,6,7,8,9), on parle alors de base décimale.

I.1 Le bit.

Bit signifie "binary digit", c'est-à-dire 0 ou 1 en numérotation binaire. C'est la plus petite unité d'information manipulable par une machine numérique. Il est possible de représenter physiquement cette information binaire :

- par un signal électrique ou magnétique, qui, lorsqu'elle atteint une certaine valeur, correspond à la valeur 1.
- grâce à des bistables, c'est-à-dire des composants électroniques qui ont deux états d'équilibre (un correspond à l'état 1, l'autre à 0)

Avec un bit il est ainsi possible d'obtenir deux états: soit 1, soit 0. 2 bits rendent possible l'obtention de quatre états différents (2*2):

2 bits
0 0
0 1
1 0
1 1

Avec 3 bits il est possible d'obtenir huit états différents (2*2*2):
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

Pour un groupe de n bits, il est possible de représenter 2^n valeurs.

I.2 L'octet.

L'octet est une unité d'information composée de 8 bits. Il permet de stocker un caractère, telle qu'une lettre, un chiffre ... Ce regroupement de nombres par série de 8 permet une lisibilité plus grande, au même titre que l'on apprécie, en base décimale, de regrouper les nombres par trois pour pouvoir distinguer les milliers. Par exemple le nombre 1 256 245 est plus lisible que 1256245. Une unité d'information composée de 16 bits est généralement appelée *mot* (en anglais *word*). Une unité d'information de 32 bits de longueur est appelée *double mot* (en anglais *double word*, d'où l'appellation *dword*). Pour un octet, le plus petit nombre est 0 (représenté par huit zéros 00000000), le plus grand est 255 (représenté par huit chiffre "un" 11111111), ce qui représente 256 possibilités de valeurs différentes.

2^7 =128	2^6 =64	2^5 =32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1

KiloOctets, MégaOctets

Longtemps l'informatique s'est singularisée par l'utilisation des unités du système international avec des valeurs différentes. Ainsi beaucoup d'informaticiens ont appris que 1kilo-ocet=1024 octets. Hors depuis décembre 1998, l'organisme international IEC a statué (http://physics.nist.gov./cuu/Units/binary.html). L'informatique utilise donc:

Un kilo-octet (Ko) = 1000 octets

Un méga-octet (Mo) = 1000 Ko = 1 000 000 octets

Un giga-octet (Go) = 1000 Mo = 1 000 000 000 octets

Un tera-octet (To) = 1000 Go = 1 000 000 000 000 octets

Comme tout le monde serais-je tenté de dire, mais également le kilo binaire (kibi), le méga binaire (mébi), le giga binaire (gibi), le tera binaire (tebi) définis comme ceci:

Un kibi-octet (Kio) vaut 2^{10} = 1024 octets
Un mébi-octet (Meo)vaut 2^{20} =1 048 576 octets
Un gibi-octet (Gio) vaut 2^{30} =1 073 741 824 octets
Un tebi-octet (Tio) vaut 2^{40} =1 099 511 627 776 octets
Il est également utile de noter que la communauté internationale dans son ensemble utilise le byte de préférence à l'octet purement francophone.

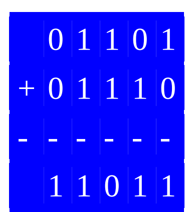
II. Les opérations en binaire.

Les opérations arithmétiques simples telles que l'addition, la soustraction et la multiplication sont faciles à effectuer en binaire.

II.1 L'addition en binaire.

L'addition en binaire se fait avec les mêmes règles qu'en décimale: On commence à additionner les bits de poids faibles (les bits de droite) puis on a des retenues lorsque la somme de deux bits de mêmes poids dépasse la valeur de l'unité la plus grande (dans le cas du binaire: 1), cette retenue est reportée sur le bit de poids plus fort suivant...

Par exemple:



	0	1	1	0	1
+	0	1	1	1	0
-	-	-	-	-	-
	1	1	0	1	1

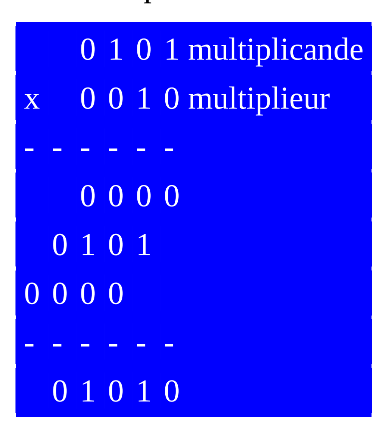
II.2 La multiplication en binaire.

La table de multiplication en binaire est très simple:

- 0x0=0
- 0x1=0
- 1x0=0
- 1x1=1

La multiplication se fait en formant un produit partiel pour chaque digit du multiplieur (seul les bits non nuls donneront un résultat non nul). Lorsque le bit du multiplieur est nul, le produit par l est nul, lorsqu'il vaut un, le produit partiel est constitué du multiplicande décalé du nombre de positions égal au poids du bit du multiplieur.

Par exemple:



		0	1	0	1	multiplicande
x		0	0	1	0	multiplieur
-	-	-	-	-	-	
		0	0	0	0	
	0	1	0	1		
0	0	0	0			
-	-	-	-	-	-	
	0	1	0	1	0	

III. La base hexadécimale.

Les nombres binaires étant de plus en plus longs, il a fallu introduire une nouvelle base: la base hexadécimale. La base hexadécimale consiste à compter sur une base 16, c'est pourquoi

au-delà des 10 premiers chiffres on a décidé d'ajouter les 6 premières lettres : 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

Base décimale	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base hexa –	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Base binaire	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Un exemple :

Le nombre 27 (en base 10) vaut en base 16 : $16+11=1*16^1 + 11*16^0= 1*16^1 + B*16^0$ c'est-à-dire 1B en base 16. Le nombre FB3 (en base 16) vaut en base 10 : $F*16^2 + B*16^1 + 3*16^0=3840+176+3=4019$. Pour convertir un octet en hexadécimale, on le partage en 2 groupes de 4 bits, qui correspondent chacun à un chiffre hexadécimal.

2	A	D	5
0010	1010	1101	0101

IV. Représentation d'un nombre dans un ordinateur.

On appelle représentation (ou codification) d'un nombre la façon selon laquelle il est décrit sous forme binaire. La représentation des nombres sur un ordinateur est indispensable pour que celui-ci puisse les stocker et les manipuler. Toutefois le problème est qu'un nombre mathématique peut être infini (aussi grand que l'on veut), mais la représentation d'un nombre dans un ordinateur doit être fait sur un nombre de bits prédéfini. Il s'agit donc de prédéfinir un nombre de bits et la manière de les utiliser pour que ceux-ci servent le plus efficacement possible à représenter l'entité.

IV.1 Représentation d'un entier naturel.

Un entier naturel est un entier positif ou nul. Le choix à faire (c'est-à-dire le nombre de bits à utiliser) dépend de la fourchette des nombres que l'on désire utiliser. Pour coder des nombres entiers naturels compris entre 0 et 255, il nous suffira de 8 bits (un octet) car $2^8=256$. D'une manière générale un codage sur *n* bits pourra permettre de représenter des nombres entiers naturels compris entre 0 et 2^n-1. Pour représenter un nombre entier naturel, après avoir défini le nombre de bits sur lequel on le code, il suffit de ranger chaque bit dans la cellule binaire correspondant à son poids binaire de la droite vers la gauche, puis on "remplit" les bits non utilisés par des zéros.

IV.2 Représentation d'un entier signé.

Un entier signé est un entier pouvant être négatif. Il faut donc coder le nombre de telle façon que l'on puisse savoir s'il s'agit d'un nombre positif ou d'un nombre négatif, et il faut de plus que les règles d'addition soient conservées. L'astuce consiste à utiliser un codage que l'on appelle *complément à deux*.

- un entier relatif positif ou nul sera représenté en binaire (base 2) comme un entier naturel, à la seule différence que le bit de poids fort (le bit situé à l'extrême gauche) représente le signe. Il faut donc s'assurer pour un entier positif ou nul qu'il est à zéro (0 correspond à un signe positif, 1 à un signe négatif). Ainsi si on code un entier naturel sur 4 bits, le nombre le plus grand sera 0111 (c'est-à-dire 7 en base décimale). D'une manière générale le plus grand entier relatif positif codé sur *n* bits sera $2^{n-1}-1$.

-un entier relatif négatif grâce au codage en complément à deux. Soit à représenter un nombre négatif.

- Prenons son opposé (son équivalent en positif)

- On le représente en base 2 sur n-1 bits

- On complémente chaque bit (on inverse, c'est-à-dire que l'on remplace les zéros par des 1 et vice-versa)

- On ajoute 1

On remarquera qu'en ajoutant le nombre et son complément à deux on obtient 0. Voyons maintenant cela sur un exemple: On désire coder la valeur -5 sur 8 bits. Il suffit :

- d'écrire 5 en binaire: 00000101

- de complémenter à 1: 11111010

- d'ajouter 1: 11111011

- la représentation binaire de -5 sur 8 bits est 11111011

Remarques:

Le bit de poids fort est 1, on a donc bien un nombre négatif. Si on ajoute 5 et -5 (00000101 et 11111011) on obtient 0 (avec une retenue de 1...)

IV.3 Représentation d'un nombre réel.

Il s'agit d'aller représenter un nombre binaire à virgule (par exemple *101,01* qui ne se lit pas *cent un virgule zéro un* puisque c'est un nombre binaire mais *5,25* en décimale) sous la forme 1,XXXXX... * 2^n (c'est-à-dire dans notre exemple 1,0101*2^2). La norme *IEEE* définit la façon de coder un nombre réel. Cette norme se propose de coder le nombre sur 32 bits et définit trois composantes:

- le signe est représenté par un seul bit, le bit de poids fort (celui le plus à gauche) ;
- l'exposant est codé sur les 8 bits consécutifs au signe ;
- la mantisse (les bits situés après la virgule) sur les 23 bits restants ;

Ainsi le codage se fait sous la forme suivante:

seeeeeeeemmmmmmmmmmmmmmmmmmmmmmm

- le **s** représente le bit relatif au signe
- les **e** représentent les bits relatifs à l'exposant
- les **m** représentent les bits relatifs à la mantisse

Certaines conditions sont toutefois à respecter pour les exposants:

- l'exposant 00000000 est interdit
- l'exposant 11111111 est interdit. On s'en sert toutefois pour signaler des erreurs, on appelle alors cette configuration du nombre *NaN*, ce qui signifie *Not a number*
- les exposants peuvent ainsi aller de -126 à 127

Voyons voir ce codage sur un exemple. Soit à coder la valeur 525,5 :

- 525,5 s'écrit en base 2 de la façon suivante: 1000001101,1. On veut l'écrire sous la forme :

1,0000011011 x 2^9 . Par conséquent, le bit s vaut 1, l'exposant vaut 9 : soit 1001 et la mantisse est 10000011011. La représentation du nombre 525.5 en binaire avec la norme IEEE est: 1000010010000000000000010000011011

I. Mise en situation.

Un système numérique, intégrant de l'électronique, fait souvent apparaître des fonctions ayant pour rôle le traitement d'informations : opérations arithmétiques (addition, multiplication...) ou logiques (ET, OU...) entre plusieurs signaux d'entrée permettant de générer des signaux de sortie. Ces fonctions peuvent être réalisées par des circuits intégrés analogiques ou logiques. Mais, lorsque le système devient complexe, et qu'il est alors nécessaire de réaliser un ensemble important de traitements d'informations, il devient plus simple de faire appel à une structure à base de microcontrôleur ou microprocesseur. Le développement de ces composants programmables a été rendu possible grâce l'essor considérable qu'a connu la microélectronique et notamment les techniques d'intégration. Cette évolution a permis en 1971, la fabrication du premier microprocesseur par la société INTEL. Ce microprocesseur, le « 4004 », comportait déjà 2300 transistors et fonctionnait avec un bus de données de 4 bits. Depuis, l'intégration du nombre de transistors dans les microprocesseurs n'a cessé d'évoluer, parallèlement à la puissance de calcul et la rapidité d'exécution. Aujourd'hui, un microprocesseur Pentium IV comporte a peu près 24 millions de transistors et peut traiter des données de 8, 16, 32, 64 bits en même temps. La puissance des microprocesseurs d'aujourd'hui a orientée leur utilisations vers le traitement des informations de masse (Gestion d'une base de donnée, Gestion des périphériques bloc, …), le calcul scientifique ainsi que tout ce qui est interface homme machine réactif (clavier, souris, écran, …). Comme nous pouvons le constater, le domaine d'application des microprocesseurs reste vaste. C'est pourquoi nous les classons dans la catégorie des composants programmables généralistes, cela signifie qu'ils peuvent tout faire, mais ils ne sont optimisés pour rien. La majorité des microprocesseurs ont une architecture CISC (*Complex Instruction Set Computer*, ce qui signifie "ordinateur avec jeu d'instructions complexes". C'est le cas des processeurs de type x86, c'est-à-dire les processeurs fabriqués par Intel, AMD, Cyrix, ... Les processeurs basés sur l'architecture CISC peuvent traiter des instructions complexes, qui sont directement câblées sur leurs circuits électroniques, c'est-à-dire que certaines instructions difficiles à créer à partir des instructions de base sont directement imprimées sur le silicium de la puce afin de gagner en rapidité d'exécution. L'inconvénient de ce type d'architecture provient justement du fait que des fonctions supplémentaires sont imprimées sur le silicium, d'où un coût élevé. D'autre part, les instructions sont de longueurs variables et peuvent parfois prendre plus d'un cycle d'horloge ce qui les rend lentes à l'exécution. Néanmoins, avec la considérable augmentation de la taille des puces électroniques et la gigantesque accélération des fréquences d'horloge, la

puissance de calcul d’un microprocesseur CISC d’aujourd’hui est considérable. Son caractère généraliste lui permet d’être par excellence le composant de base de l’informatique. Mais en instrumentation et automatisme on lui préférera généralement des composants plus spécialisés ne nécessitant pas, ni un calcul complexe ni un traitement d’informations de masse. C’est pourquoi dans les applications industrielles, que ce soit d’automatisme ou d’instrumentation, le microcontrôleur est le composant programmable le plus utilisé. Il comporte sur sa puce un certain nombre d'interfaces qui n'existent pas sur un microprocesseur, par contre il est généralement moins puissant en terme de rapidité ou de taille de mémoire adressable et le plus souvent cantonné aux données de 8 ou 16 bits. Les microcontrôleurs utilisent la technologie RISC (Reduced Instruction Set Computer), dont la traduction est "ordinateur à jeu d'instructions réduit" et n'a pas de fonctions supplémentaires câblées. Ce qui qui implique une programmation plus difficile et un compilateur plus puissant. Les instructions d’un microcontrôleur sont tellement peu nombreuses (en moyenne une soixantaine) qu'il est possible de les graver directement sur le silicium sans alourdir de manière dramatique leur fabrication. L'avantage d'une telle architecture est bien évidemment le coût réduit au niveau de la fabrication des processeurs l'utilisant. De plus, les instructions, étant simples, ils sont exécutés en un cycle d'horloge, ce qui rend l'exécution des programmes plus rapides qu'avec des processeurs basés sur une architecture CISC. En plus, de tels processeurs sont capables de traiter plusieurs instructions simultanément en les traitant en parallèle. Les microcontrôleurs ont permis de faire évoluer les systèmes micro programmés vers encore plus de simplicité et de rapidité. Ils sont aujourd’hui utilisés dans la plupart des réalisations industrielles grand public ou professionnelles, ils gèrent au plus juste et au plus vite les applications. Leur évolution a permis l'intégration de circuits complexes variés. Ces circuits ont été intégrés sur une même puce donnant ainsi beaucoup de flexibilité et de puissance de commande au microcontrôleur. Cette polyvalence lui permet d'occuper une place importante que ce soit en instrumentation, en commande ou en automatisme industriel. Le meilleur exemple est bien évidemment les automates programmables qui sont tous équipés de microcontrôleurs.

II. Description et structure interne d’un microcontrôleur.

Un microcontrôleur est un composant réunissant sur un seul et même silicium un micro-processeur, divers dispositifs d'entrées/sorties et de contrôle d'interruptions ainsi que de la mémoire, notamment pour stocker le programme d’application. Dédié au contrôle, il embarque également un certain nombre de périphériques spécifiques

des domaines ciblés (bus série, interface parallèle, convertisseur analogique numérique, ...). Les microcontrôleurs améliorent l'intégration et le coût (lié à la conception et à la réalisation) d'un système à base de microprocesseur en rassemblant ces éléments essentiels dans un seul circuit intégré. On parle alors de "système sur une puce" (en anglais : "System On chip"). Il existe plusieurs familles de microcontrôleurs, se différenciant par la vitesse de leur processeur et par le nombre de périphériques qui les composent. Toutes ces familles ont un point commun c'est de réunir tous les éléments essentiels d'une structure à base de microprocesseur sur une même puce. Voici généralement ce que l'on trouve à l'intérieur d'un tel composant :

☞ Un microprocesseur (C.P.U.),

☞ Des bus,

☞ De la mémoire de donnée (RAM et EEPROM),

☞ De la mémoire programme (ROM, OTPROM, UVPROM ou EEPROM),

☞ Des interfaces parallèles pour la connexion des entrées / sorties,

☞ Des interfaces séries (synchrone ou asynchrone) pour le dialogue avec d'autres unités,

☞ Des timers pour générer ou mesurer des signaux avec une grande précision temporelle.

Sur la figure (2.1) nous présentons le schéma type d'un microcontrôleur.

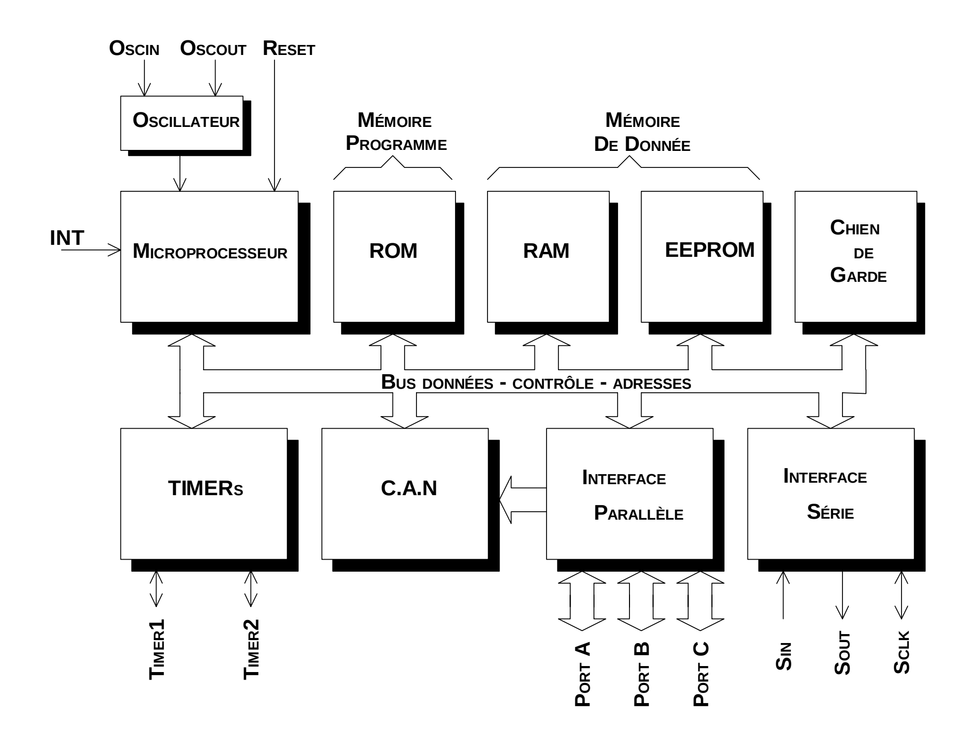


Figure 2.1. Schéma type de tout appareil programmable.

La présence de ces divers éléments de base est indispensable et, même s'ils ne sont pas toujours aussi visibles que sur notre synoptique, ils sont toujours présents. Ces éléments sont les mêmes que pour un système informatique classique mais, dans le cadre d'une application pouvant être traitée par un microcontrôleur, leurs tailles et fonctions différent un peu de ce que nous avons peut-être l'habitude de voir. L'unité centrale, généralement constituée par un microprocesseur plus ou moins évolué, exécute le programme qui va donner vie à l'application. Pour les applications industrielles, ces programmes ont en commun le fait qu'ils ne nécessitent que très rarement des calculs complexes alors qu'ils sont très friands de manipulations d'informations d'entrées/sorties. C'est pour cette raison que la majorité des microcontrôleurs inclut une interface parallèle de type PIA(Peripheral Interface Adapter figure (2.1)). Etant donné la nature des applications des microcontrôleurs souvent les programmes sont contenus dans le deuxième élément de la figure (2.1) qui est la mémoire morte ou ROM. Cette mémoire peut-être constituée de diverses façons: mémoire programmée par masque, mémoire UVPROM ou EEPROM. De cette façon, nous n'avons plus besoin de charger le programme d'application à chaque mise sous tension. L'exécution de l'application est enclenchée

automatiquement à la mise sous tension, ce qui n'est pas le cas pour les systèmes à base de microprocesseur qui nécessitent un programme de démarrage (système d'exploitation ou moniteur). Pour pouvoir travailler correctement notre microprocesseur a souvent besoin de stocker des données temporaires quelque part et c'est là qu'intervient la mémoire vive ou RAM qui, contrairement aux systèmes informatiques classiques, peut être de très petite taille. Le dernier élément qui ne manque pas d'importance dans les applications susceptible de faire appel à un microcontrôleur est tout ce qui concerne les circuits d'interface avec le monde extérieur. Contrairement aux systèmes informatiques classiques où les interfaces sont bien connues, de types assez peu diversifiés et en nombre relativement limité; les interfaces que nous pouvons rencontrer dans des applications industrielles sont absolument quelconques. Il y a en effet assez peu de points communs entre un moteur pas à pas, un afficheur à cristaux liquides ou bien encore un programmateur de machine à laver. Le schéma type d'un microcontrôleur est issu d'une analyse assez forte des divers systèmes de commande et d'automatisme réalisés avant l'avènement des microcontrôleurs. Les fabricants de circuits intégrés ont affinés un peu la définition de ce qu'il fallait intégrer pour arriver à un schéma-type analogue à celui de la figure (1.1). Nous y retrouvons bien évidemment un microprocesseur, généralement simplifié et avec un jeu d'instructions réduit mais auquel des instructions de manipulation de bits, très utiles pour faire des entrées/sorties, lui ont été ajoutées. Dans certains cas, ce microprocesseur se voit doté d'un très grand nombre de registres internes qui servent alors de mémoire vive. Ce qui évite l'utilisation d'une mémoire externe pour le stockage des données et rend le système encore plus compact. Dans un grand nombre de microcontrôleurs, nous trouvons également de la mémoire morte intégré. Avec les progrès technologiques les fabricants ont appris à placer sur la puce de la mémoire programmable électriquement et effaçable aux ultraviolets (UVPROM) ou, plus récemment encore, de la mémoire programmable et effaçable électriquement (EEPROM). On trouve donc à l'heure actuelle au moins quatre types différents de microcontrôleurs :

- ceux sans aucune mémoire morte ;

- ceux avec EPROM (programmable et effaçable à l'ultra violet);

-ceux avec EEPROM ;

-ceux avec un mélange de ces combinaisons (ROM -EEPROM ou UVPROM -EEPROM).

Pour ce qui est de la mémoire vive ou RAM, la situation est plus simple. Quasiment tous les microcontrôleurs disposent d'une RAM interne de taille en principe assez faible et, lorsqu'elle n'est pas explicitement visible sur le synoptique, c'est que l'unité centrale dispose d'assez de registres pour servir de RAM comme nous l'avons expliqué précédemment. Il est un peu plus délicat de faire un schéma type au niveau des circuits d'interface car c'est là que les différents microcontrôleurs se distinguent en fonction des créneaux d'applications qu'ils visent. Néanmoins, on rencontre généralement les éléments de base suivants :

- des lignes d'entrées/sorties parallèles en nombre variable selon la vocation et la taille du boîtier (un problème de nombre maximum de pattes se posant très vite avec l'accroissement du nombre de ces lignes);

- au moins une interface d'entrée/sortie série asynchrone, plus ou moins évoluée selon les circuits;

- un ou plusieurs timers internes dont les possibilités peuvent être très variables mais qui fonctionnent généralement en compteurs, décompteurs, générateurs d'impulsions programmables, etc. ;

- parfois, mais c'est un peu plus rare, un ou des convertisseurs analogiques numériques précédés ou non de multiplexeurs pour offrir plusieurs voies ;

- parfois aussi, mais c'est également plus rare, un convertisseur numérique analogique.

Enfin, bien que ce ne soit pas une véritable interface d'entrée/sortie au sens où nous l'entendons, certains microcontrôleurs disposent d'un accès à leur bus interne. Ceci permet de connecter des boîtiers destinés à accomplir des fonctions qui font défaut sur la puce ce qui est parfois utile. Précisons, mais c'est une évidence, que tous les microcontrôleurs sans mémoire morte interne disposent nécessairement de cette interface puisqu'il faut impérativement leur permettre d'accéder à une mémoire morte externe contenant le programme de l’application. Bien sûr, notre schéma-type ne peut recouvrir tous les cas mais il vous donne une idée assez précise de ce que contient un microcontrôleur. Il est bien évident d’après ce schéma que la puissance d’un microcontrôleur est directement liée au processeur qu’il intègre. C'est

pourquoi les constructeurs développent souvent un cœur de processeur destiné aussi bien à décliner une gamme de microprocesseurs que de microcontrôleurs ; La tendance aujourd'hui et un cœur de processeur 16 ou 32 bits qui représente une augmentation de la surface occupée sur le silicium de seulement quelques pour-cent par rapport à un circuit en 8 bits. Le prix du microcontrôleur étant directement lié à sa taille, embarquer une puissance supérieure est un choix qui ne grève plus le budget. Opter pour un cœur de processeur en 16 ou en 32 bits, c'est permettre entre autres des exécutions parallèles, un espace mémoire élargi, des interfaces de communications ou encore le remplacement de fonctions analogiques par des traitements numériques. Le cœur de processeur 32 bits le plus prisé aujourd'hui est l'ARM7. Pour preuve, plusieurs fabricants l'utilisent aujourd'hui pour le développement de microcontrôleurs 32 bits. Les secteurs les plus visés sont surtout les systèmes de communication sans fil, les téléphones portables et l'industrie automobile. Par exemple les microcontrôleurs de la famille STR7 de STMicroelectronics sont basés sur le cœur ARM7. Ces microcontrôleurs intègrent une multitude d'interfaces série (CAN, USB, I2C, SPI,..), de la mémoire SDRAM, de la mémoire FLASH, des convertisseurs analogiques numériques, 48 entrées/sorties, une interface JTAG, plusieurs horloges temps réel,.. L étude d'un microcontrôleur basé sur le cœur ARM7 comme le STR7 dépasse largement le cadre de ce cours. La diversité des périphériques qu'il intègre rend sa compréhension très difficile c'est pourquoi nous préférons présenter un cœur de processeur beaucoup plus simple comme exemple d'étude dans le cadre de ce cours. Dans le paragraphe suivant nous allons présenter les éléments généraux communs aux divers types de processeurs.

III. Le processeur.

Le processeur (microprocesseur) est le composant hardware le plus connu d'un système micro-programmé. C'est l'unité intelligente de traitement des informations. Son travail consiste à lire des programmes (des suites d'instructions), à les décoder et à les exécuter. Les années 80 voyaient l'émergence de ces circuits avec les Zylog Z80, 6800 de Motorola, le 8085 de Intel qui est souvent utilisé en tant que microcontrôleur. Avec l'arrivée des PC-XT d'IBM et l'utilisation du 8088, INTEL devenait maître du marché fin des années 80. L'interfaçage du processeur avec l'extérieur nécessite 3 bus: un bus de données, un bus d'adresse et un bus de commande. Il existe des processeurs basés sur l'architecture CISC et d'autres basés sur l'architecture RISC. Cependant certains processeurs sont difficilement classifiables comme le CPU i486 également appelé 80486. Ce véritable processeur 32 bits (bus internes et externes) est à mi chemin entre le tout CISC et le tout RISC. Ce processeur offre des performances similaires aux processeurs RISC 32 bits, tout en restant compatible 100% avec son

prédécesseur CISC le 80386. Par contre le processeur 68040 de Motorola est basé sur l'architecture CISC. Il est présenté par Motorola comme étant un processeur CISC pur et dur. Les premiers concepteurs de processeurs rajoutaient le plus d'instructions possibles pour permettre à l'utilisateur de peaufiner ses programmes. Néanmoins, ces multiples instructions ralentissent le fonctionnement du microprocesseur et sont peu utilisées en pratique. Actuellement, on utilise de plus en plus de processeurs **RISC** (Reduced Instruction Set Computer). Le nombre d'instructions est réduit, mais exécutées nettement plus rapidement. Chaque instruction complexe peut être programmée par plusieurs instructions simples. Un processeur est constitué de:

- une unité de commande qui lit les instructions et les décode;
- une unité de traitement (UAL - unité arithmétique et logique) qui exécute les instructions;
- d'un ensemble de mémoire appelés registres;
- d'un bus de données externe;
- d'un bus d'adresse externe;
- d'un bus de commande externe;
- d'un bus de données interne reliant l'unité de commande l'UAL et les registres.

Lorsque tous ces éléments sont regroupés sur une même puce, on parle alors de microprocesseur. La figure ci dessous donne une idée sur l'architecture interne d'un microprocesseur. Sur cette figure nous pouvons voir les 3 bus qui permettent au microprocesseur de communiquer avec l'extérieur.

III.1 Architecture de base d'un microprocesseur

III.1.1 L'unité de commande.

Elle permet de "séquencer" le déroulement des instructions. Elle effectue la recherche en mémoire de l'instruction, le décodage, l'exécution et la préparation de l'instruction suivante. L'unité de commande élabore tous les signaux de synchronisation internes ou externes (bus des commandes) au microprocesseur.

III.1.2 L'unité arithmétique et logique (UAL).

C'est l'organe qui effectue les opérations:

Arithmétiques : addition, soustraction, multiplication, ...

Logiques : et, ou, non, décalage, rotation, ...

Deux registres sont associés à l'UAL : l'accumulateur et le registre d'état.

III.1.2.1 L'accumulateur (nommé : A).

C'est une des deux entrées de l'UAL. Il est impliqué dans presque toutes les opérations réalisées par l'UAL. Certains constructeurs ont des microprocesseurs à deux accumulateurs (Motorola : 6800).

Exemple: **A** étant l'accumulateur et **B** un registre, on peut avoir : A+B (ADD A,B : addition du contenu du registre A avec celui du registre B, le résultat étant mis dans A)

I. Mise en situation.

Un système numérique, intégrant de l’électronique, fait souvent apparaître des fonctions ayant pour rôle le traitement d’informations : opérations arithmétiques (addition, multiplication...) ou logiques (ET, OU...) entre plusieurs signaux d’entrée permettant de générer des signaux de sortie. Ces fonctions peuvent être réalisées par des circuits intégrés analogiques ou logiques. Mais, lorsque le système devient complexe, et qu’il est alors nécessaire de réaliser un ensemble important de traitements d’informations, il devient plus simple de faire appel à une structure à base de microcontrôleur ou microprocesseur. Le développement de ces composants programmables a été rendu possible grâce l’essor considérable qu’a connu la microélectronique et notamment les techniques d’intégration. Cette évolution a permis en 1971, la fabrication du premier microprocesseur par la société INTEL. Ce microprocesseur, le « 4004 », comportait déjà 2300 transistors et fonctionnait avec un bus de données de 4 bits. Depuis, l’intégration du nombre de transistors dans les microprocesseurs n’a cessé d’évoluer, parallèlement à la puissance de calcul et la rapidité d’exécution. Aujourd’hui, un microprocesseur Pentium IV comporte a peu près 24 millions de transistors et peut traiter des données de 8, 16, 32, 64 bits en même temps. La puissance des microprocesseurs d’aujourd’hui a orientée leur utilisations vers le traitement des informations de masse (Gestion d’une base de donnée, Gestion des périphériques bloc, …), le calcul scientifique ainsi que tout ce qui est interface homme machine réactif (clavier, souris, écran, …). Comme nous pouvons le constater, le domaine d’application des microprocesseurs reste vaste. C’est pourquoi nous les classons dans la catégorie des composants programmables généralistes, cela signifie qu'ils peuvent tout faire, mais ils ne sont optimisés pour rien. La majorité des

microprocesseurs ont une architecture CISC (*Complex Instruction Set Computer*, ce qui signifie "ordinateur avec jeu d'instructions complexes". C'est le cas des processeurs de type x86, c'est-à-dire les processeurs fabriqués par Intel, AMD, Cyrix, ... Les processeurs basés sur l'architecture CISC peuvent traiter des instructions complexes, qui sont directement câblées sur leurs circuits électroniques, c'est-à-dire que certaines instructions difficiles à créer à partir des instructions de base sont directement imprimées sur le silicium de la puce afin de gagner en rapidité d'exécution. L'inconvénient de ce type d'architecture provient justement du fait que des fonctions supplémentaires sont imprimées sur le silicium, d'où un coût élevé. D'autre part, les instructions sont de longueurs variables et peuvent parfois prendre plus d'un cycle d'horloge ce qui les rend lentes à l'exécution. Néanmoins, avec la considérable augmentation de la taille des puces électroniques et la gigantesque accélération des fréquences d'horloge, la puissance de calcul d'un microprocesseur CISC d'aujourd'hui est considérable. Son caractère généraliste lui permet d'être par excellence le composant de base de l'informatique. Mais en instrumentation et automatisme on lui préférera généralement des composants plus spécialisés ne nécessitant pas, ni un calcul complexe ni un traitement d'informations de masse. C'est pourquoi dans les applications industrielles, que ce soit d'automatisme ou d'instrumentation, le microcontrôleur est le composant programmable le plus utilisé. Il comporte sur sa puce un certain nombre d'interfaces qui n'existent pas sur un microprocesseur, par contre il est généralement moins puissant en terme de rapidité ou de taille de mémoire adressable et le plus souvent cantonné aux données de 8 ou 16 bits. Les microcontrôleurs utilisent la technologie RISC (Reduced Instruction Set Computer), dont la traduction est "ordinateur à jeu d'instructions réduit" et n'a pas de fonctions supplémentaires câblées. Ce qui qui implique une programmation plus difficile et un compilateur plus puissant. Les instructions d'un microcontrôleur sont tellement peu nombreuses (en moyenne une soixantaine) qu'il est possible de les graver directement sur le silicium sans alourdir de manière dramatique leur fabrication. L'avantage d'une telle architecture est bien évidemment le coût réduit au niveau de la fabrication des processeurs l'utilisant. De plus, les instructions, étant simples, ils sont exécutés en un cycle d'horloge, ce qui rend l'exécution des programmes plus rapides qu'avec des processeurs basés sur une architecture CISC. En plus, de tels processeurs sont capables de traiter plusieurs instructions simultanément en les traitant en parallèle. Les microcontrôleurs ont permis de faire évoluer les systèmes micro programmés vers encore plus de simplicité et de rapidité. Ils sont aujourd'hui utilisés dans la plupart des réalisations industrielles grand public ou professionnelles, ils gèrent au plus juste et au plus vite les applications. Leur évolution a permis l'intégration de circuits complexes variés. Ces

circuits ont été intégrés sur une même puce donnant ainsi beaucoup de flexibilité et de puissance de commande au microcontrôleur. Cette polyvalence lui permet d'occuper une place importante que ce soit en instrumentation, en commande ou en automatisme industriel. Le meilleur exemple est bien évidemment les automates programmables qui sont tous équipés de microcontrôleurs.

II. Description et structure interne d'un microcontrôleur.

Un microcontrôleur est un composant réunissant sur un seul et même silicium un micro-processeur, divers dispositifs d'entrées/sorties et de contrôle d'interruptions ainsi que de la mémoire, notamment pour stocker le programme d'application. Dédié au contrôle, il embarque également un certain nombre de périphériques spécifiques des domaines ciblés (bus série, interface parallèle, convertisseur analogique numérique, ...). Les microcontrôleurs améliorent l'intégration et le coût (lié à la conception et à la réalisation) d'un système à base de microprocesseur en rassemblant ces éléments essentiels dans un seul circuit intégré. On parle alors de "système sur une puce" (en anglais : "System On chip"). Il existe plusieurs familles de microcontrôleurs, se différenciant par la vitesse de leur processeur et par le nombre de périphériques qui les composent. Toutes ces familles ont un point commun c'est de réunir tous les éléments essentiels d'une structure à base de microprocesseur sur une même puce. Voici généralement ce que l'on trouve à l'intérieur d'un tel composant :

☞ Un microprocesseur (C.P.U.),

☞ Des bus,

☞ De la mémoire de donnée (RAM et EEPROM),

☞ De la mémoire programme (ROM, OTPROM, UVPROM ou EEPROM),

☞ Des interfaces parallèles pour la connexion des entrées / sorties,

☞ Des interfaces séries (synchrone ou asynchrone) pour le dialogue avec d'autres unités,

☞ Des timers pour générer ou mesurer des signaux avec une grande précision temporelle.

Sur la figure (2.1) nous présentons le schéma type d'un microcontrôleur.

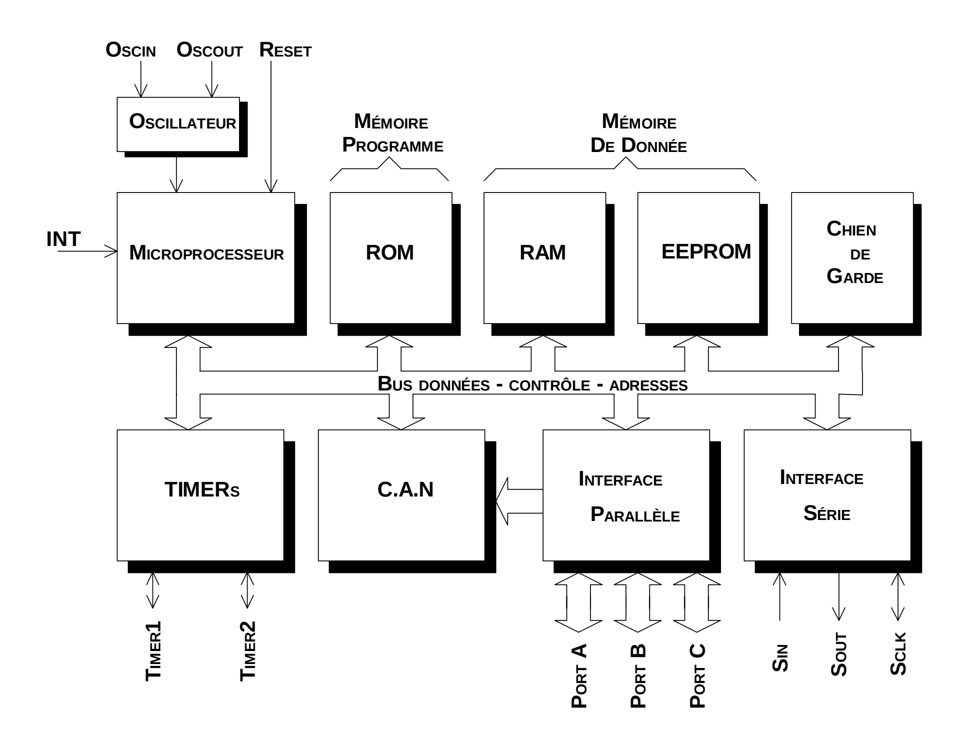


Figure 2.1. Schéma type de tout appareil programmable.

La présence de ces divers éléments de base est indispensable et, même s'ils ne sont pas toujours aussi visibles que sur notre synoptique, ils sont toujours présents. Ces éléments sont les mêmes que pour un système informatique classique mais, dans le cadre d'une application pouvant être traitée par un microcontrôleur, leurs tailles et fonctions différent un peu de ce que nous avons peut-être l'habitude de voir. L'unité centrale, généralement constituée par un microprocesseur plus ou moins évolué, exécute le programme qui va donner vie à l'application. Pour les applications industrielles, ces programmes ont en commun le fait qu'ils ne nécessitent que très rarement des calculs complexes alors qu'ils sont très friands de manipulations d'informations d'entrées/sorties. C'est pour cette raison que la majorité des microcontrôleurs inclut une interface parallèle de type PIA(Peripheral Interface Adapter figure (2.1)). Etant donné la nature des applications des microcontrôleurs souvent les programmes sont contenus dans le deuxième élément de la figure (2.1) qui est la mémoire morte ou ROM. Cette mémoire peut-être constituée de diverses

façons: mémoire programmée par masque, mémoire UVPROM ou EEPROM. De cette façon, nous n'avons plus besoin de charger le programme d'application à chaque mise sous tension. L'exécution de l'application est enclenchée automatiquement à la mise sous tension, ce qui n'est pas le cas pour les systèmes à base de microprocesseur qui nécessitent un programme de démarrage (système d'exploitation ou moniteur). Pour pouvoir travailler correctement notre microprocesseur a souvent besoin de stocker des données temporaires quelque part et c'est là qu'intervient la mémoire vive ou RAM qui, contrairement aux systèmes informatiques classiques, peut être de très petite taille. Le dernier élément qui ne manque pas d'importance dans les applications susceptible de faire appel à un microcontrôleur est tout ce qui concerne les circuits d'interface avec le monde extérieur. Contrairement aux systèmes informatiques classiques où les interfaces sont bien connues, de types assez peu diversifiés et en nombre relativement limité; les interfaces que nous pouvons rencontrer dans des applications industrielles sont absolument quelconques. Il y a en effet assez peu de points communs entre un moteur pas à pas, un afficheur à cristaux liquides ou bien encore un programmateur de machine à laver. Le schéma type d'un microcontrôleur est issu d'une analyse assez forte des divers systèmes de commande et d'automatisme réalisés avant l'avènement des microcontrôleurs. Les fabricants de circuits intégrés ont affinés un peu la définition de ce qu'il fallait intégrer pour arriver à un schéma-type analogue à celui de la figure (1.1). Nous y retrouvons bien évidemment un microprocesseur, généralement simplifié et avec un jeu d'instructions réduit mais auquel des instructions de manipulation de bits, très utiles pour faire des entrées/sorties, lui ont été ajoutées. Dans certains cas, ce microprocesseur se voit doté d'un très grand nombre de registres internes qui servent alors de mémoire vive. Ce qui évite l'utilisation d'une mémoire externe pour le stockage des données et rend le système encore plus compact. Dans un grand nombre de microcontrôleurs, nous trouvons également de la mémoire morte intégré. Avec les progrès technologiques les fabricants ont appris à placer sur la puce de la mémoire programmable électriquement et effaçable aux ultraviolets (UVPROM) ou, plus récemment encore, de la mémoire programmable et effaçable électriquement (EEPROM). On trouve donc à l'heure actuelle au moins quatre types différents de microcontrôleurs :

- ceux sans aucune mémoire morte ;

- ceux avec EPROM (programmable et effaçable à l'ultra violet);

-ceux avec EEPROM ;

-ceux avec un mélange de ces combinaisons (ROM -EEPROM ou UVPROM - EEPROM).

Pour ce qui est de la mémoire vive ou RAM, la situation est plus simple. Quasiment tous les microcontrôleurs disposent d'une RAM interne de taille en principe assez faible et, lorsqu'elle n'est pas explicitement visible sur le synoptique, c'est que l'unité centrale dispose d'assez de registres pour servir de RAM comme nous l'avons expliqué précédemment. Il est un peu plus délicat de faire un schéma type au niveau des circuits d'interface car c'est là que les différents microcontrôleurs se distinguent en fonction des créneaux d'applications qu'ils visent. Néanmoins, on rencontre généralement les éléments de base suivants :

- des lignes d'entrées/sorties parallèles en nombre variable selon la vocation et la taille du boîtier (un problème de nombre maximum de pattes se posant très vite avec l'accroissement du nombre de ces lignes);

- au moins une interface d'entrée/sortie série asynchrone, plus ou moins évoluée selon les circuits;

- un ou plusieurs timers internes dont les possibilités peuvent être très variables mais qui fonctionnent généralement en compteurs, décompteurs, générateurs d'impulsions programmables, etc. ;

- parfois, mais c'est un peu plus rare, un ou des convertisseurs analogiques numériques précédés ou non de multiplexeurs pour offrir plusieurs voies ;

- parfois aussi, mais c'est également plus rare, un convertisseur numérique analogique.

Enfin, bien que ce ne soit pas une véritable interface d'entrée/sortie au sens où nous l'entendons, certains microcontrôleurs disposent d'un accès à leur bus interne. Ceci permet de connecter des boîtiers destinés à accomplir des fonctions qui font défaut sur la puce ce qui est parfois utile. Précisons, mais c'est une évidence, que tous les microcontrôleurs sans mémoire morte interne disposent nécessairement de cette interface puisqu'il faut impérativement leur permettre d'accéder à une mémoire morte externe contenant le programme de l'application.

Bien sûr, notre schéma-type ne peut recouvrir tous les cas mais il vous donne une idée assez précise de ce que contient un microcontrôleur. Il est bien évident d’après ce schéma que la puissance d’un microcontrôleur est directement liée au processeur qu’il intègre. C'est pourquoi les constructeurs développent souvent un cœur de processeur destiné aussi bien à décliner une gamme de microprocesseurs que de microcontrôleurs ; La tendance aujourd’hui et un cœur de processeur 16 ou 32 bits qui représente une augmentation de la surface occupée sur le silicium de seulement quelques pour-cent par rapport à un circuit en 8 bits. Le prix du microcontrôleur étant directement lié à sa taille, embarquer une puissance supérieure est un choix qui ne grève plus le budget. Opter pour un cœur de processeur en 16 ou en 32 bits, c'est permettre entre autres des exécutions parallèles, un espace mémoire élargi, des interfaces de communications ou encore le remplacement de fonctions analogiques par des traitements numériques. Le cœur de processeur 32 bits le plus prisé aujourd'hui est l'ARM7. Pour preuve, plusieurs fabricants l’utilisent aujourd’hui pour le développement de microcontrôleurs 32 bits. Les secteurs les plus visés sont surtout les systèmes de communication sans fil, les téléphones portables et l’industrie automobile. Par exemple les microcontrôleurs de la famille STR7 de STMicroelectronics sont basés sur le cœur ARM7. Ces microcontrôleurs intègrent une multitude d’interfaces série (CAN, USB, I2C, SPI,..), de la mémoire SDRAM, de la mémoire FLASH, des convertisseurs analogiques numériques, 48 entrées/sorties, une interface JTAG, plusieurs horloges temps réel,.. L étude d’un microcontrôleur basé sur le cœur ARM7 comme le STR7 dépasse largement le cadre de ce cours. La diversité des périphériques qu’il intègre rend sa compréhension très difficile c’est pourquoi nous préférons présenter un cœur de processeur beaucoup plus simple comme exemple d’étude dans le cadre de ce cours. Dans le paragraphe suivant nous allons présenter les éléments généraux communs aux divers types de processeurs.

III. Le processeur.

Le processeur (microprocesseur) est le composant hardware le plus connu d'un système micro-programmé. C'est l'unité intelligente de traitement des informations. Son travail consiste à lire des programmes (des suites d'instructions), à les décoder et à les exécuter. Les années 80 voyaient l'émergence de ces circuits avec les Zylog Z80, 6800 de Motorola, le 8085 de Intel qui est souvent utilisé en tant que microcontrôleur. Avec l'arrivée des PC-XT d'IBM et l'utilisation du 8088, INTEL devenait maître du marché fin des années 80. L'interfaçage du processeur avec l’extérieur nécessite 3 bus: un bus de données, un bus d'adresse et un bus de commande. Il existe des processeurs basés sur l'architecture CISC et d’autres basés sur l’architecture RISC. Cependant certains processeurs sont difficilement classifiables comme le

CPU i486 également appelé 80486. Ce véritable processeur 32 bits (bus internes et externes) est à mi chemin entre le tout CISC et le tout RISC. Ce processeur offre des performances similaires aux processeurs RISC 32 bits, tout en restant compatible 100% avec son prédécesseur CISC le 80386. Par contre le processeur 68040 de Motorola est basé sur l'architecture CISC. Il est présenté par Motorola comme étant un processeur CISC pur et dur. Les premiers concepteurs de processeurs rajoutaient le plus d'instructions possibles pour permettre à l'utilisateur de peaufiner ses programmes. Néanmoins, ces multiples instructions ralentissent le fonctionnement du microprocesseur et sont peu utilisées en pratique. Actuellement, on utilise de plus en plus de processeurs **RISC** (Reduced Instruction Set Computer). Le nombre d'instructions est réduit, mais exécutées nettement plus rapidement. Chaque instruction complexe peut être programmée par plusieurs instructions simples. Un processeur est constitué de:

- une unité de commande qui lit les instructions et les décode;
- une unité de traitement (UAL - unité arithmétique et logique) qui exécute les instructions;
- d'un ensemble de mémoire appelés registres;
- d'un bus de données externe;
- d'un bus d'adresse externe;
- d'un bus de commande externe;
- d'un bus de données interne reliant l'unité de commande l'UAL et les registres.

Lorsque tous ces éléments sont regroupés sur une même puce, on parle alors de microprocesseur. La figure ci dessous donne une idée sur l'architecture interne d'un microprocesseur. Sur cette figure nous pouvons voir les 3 bus qui permettent au microprocesseur de communiquer avec l'extérieur.

III.1 Architecture de base d'un microprocesseur

III.1.1 L'unité de commande.

Elle permet de "séquencer" le déroulement des instructions. Elle effectue la recherche en mémoire de l'instruction, le décodage, l'exécution et la préparation de l'instruction suivante. L'unité de commande élabore tous les signaux de synchronisation internes ou externes (bus des commandes) au microprocesseur.

III.1.2 L'unité arithmétique et logique (UAL).

C'est l'organe qui effectue les opérations:

Arithmétiques : addition, soustraction, multiplication, ...

Logiques : et, ou, non, décalage, rotation, ...

Deux registres sont associés à l'UAL : l'accumulateur et le registre d'état.

III.1.2.1 L'accumulateur (nommé : A).

C'est une des deux entrées de l'UAL. Il est impliqué dans presque toutes les opérations réalisées par l'UAL. Certains constructeurs ont des microprocesseurs à deux accumulateurs (Motorola : 6800).

Exemple: **A** étant l'accumulateur et **B** un registre, on peut avoir : A+B (ADD A,B : addition du contenu du registre A avec celui du registre B, le résultat étant mis dans A)

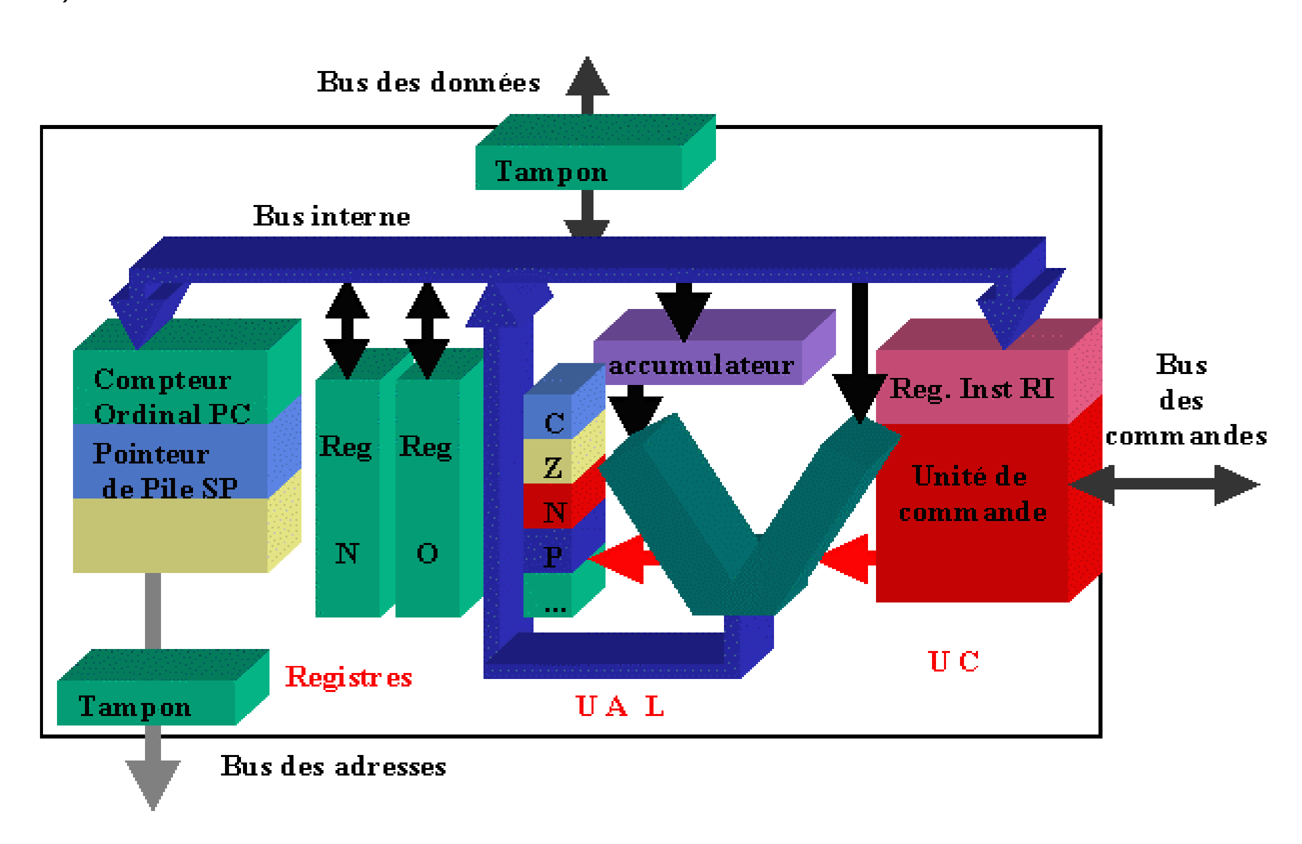


Figure 2.2. Architecture de base de microprocesseur.

III.1.2.2 Le registre d'état (Flags : F)

A chaque opération, le microprocesseur positionne un certain nombre de bascules d'état. Ces bascules sont appelées aussi indicateurs d'état ou drapeaux (status, flags). Par exemple, si une soustraction donne un résultat nul, l'indicateur de zéro (Z) sera mis à **1**. Ces bascules sont regroupées dans le registre d'état

On peut citer comme indicateurs:

- retenue (carry : C)
- retenue intermédiaire (Auxiliary-Carry : AC)
- signe (Sign : S)
- débordement (Overflow : O)
- zéro (Z)
- parité (Parity : P)

III.1.2.2.1 Retenue : (carry : C).

Exemple: addition de nombres binaire sur 8 bits

		11111100			**FCH**
	+	**10000010**		+	**82H**
carry : **1**	=	**01111110**	carry : **1**	=	**7EH**

Sur cet exemple, nous pouvons remarquer que le résultat sur 8 bits est faux (7EH). Toutefois le bit de retenu expulsé n'est pas définitivement perdu. Ce bit C (carry) est stocké dans le registre d'état est peut être testé par l'utilisateur. De la même manière lors d'une opération de décalage ou de rotation ce bit peut être positionné en cas de débordement. Par exemple, soit le décalage à gauche sur d'un bit de cet octet : 10010110, après le décalage nous avons 00101100. En ce qui concerne le carry, il va être positionné à **1**, puisque le bit expulsé est égal à **1**.

III.1.2.2.2 Retenue intermédiaire : (Auxiliary Carry : AC).

Sur les opérations arithmétiques, ce bit signale une retenue entre groupes de 4 bits (Half-byte: demi-octet) d'une quantité de 8 bits.

III.1.2.2.3 Signe: (S)

Ce bit est mise à **1** lorsque le résultat de l'opération est négatif (MSB: bit de plus fort poids du résultat: à **1**).

III.1.2.2.4 Débordement : (overflow : O)

Cet indicateur est mis **1**, lorsqu'il y a un dépassement de capacité pour les opérations arithmétiques en complément à 2. Sur 8 bits, on peut coder de -128 (1000 0000) à +127 (0111 1111).

104	**0110 1000**	**- 18**	**1110 1110**
+ **26**	**+ 0001 1010**	**- 118**	**1000 1010**
=130	**= 1000 0010 (-126)**	**-136**	**0111 1000 (120) avec C=1**

Dans cet exemple et dans les deux cas de figure (opération à droite ou opération à gauche), le bit O est positionné à **1**. L'indicateur de débordement est une fonction logique (OU exclusif) de la retenue (C) et du signe (S).

III.1.2.2.5 Le bit Zéro : (Zéro : Z)

Ce bit est mis à **1** lorsque le résultat de l'opération est nul.

III.1.2.2.6 Le bit de parité : (P)

Ce bit est mis à 1 lorsque le nombre de 1 de l'accumulateur est pair.

Remarque : La plupart des instructions modifient le **registre d'état.**

Exemple :

ADD A, B positionne les drapeaux : **O, S, Z, A, P, C**

OR B, C (B ou C -> B) positionne **S, Z, P** tandis que

MOV A, B (Move, Transférer le contenu de **B** dans **A**) n'en positionne aucun.

III.1.3 Les registres.

Il y'a deux type de registres : les registres d'usage général, et les registres d'adresses (pointeurs).

III.1.3.1 Les registres d'usage général.

Ce sont des mémoires rapides, à l'intérieur du microprocesseur, qui permettent à l'UAL de manipuler des données à vitesse élevée. Ils sont connectés au bus de données interne au microprocesseur. L'adresse d'un registre est associée à son nom (on donne généralement comme nom une lettre) A, B,C...

Exemple : **MOV C,B**: transfert du contenu du registre **B** dans le registre **C**.

III.1.3.2 Les registres d'adresses (pointeurs).

Ce sont des registres connectés sur le bus d'adresses. On peut citer comme registre:

- Le compteur ordinal (pointeur de programme PC) ;
- Le pointeur de pile (stack pointer SP) ;
- Les registres d'index (index source SI et index destination DI).

III.1.3.2.1 Le compteur ordinal (pointeur de programme PC.)

Il contient l'adresse de l'instruction à rechercher en mémoire. L'unité de commande incrémente le compteur ordinal (PC) du nombre d'octets sur lequel l'instruction, en cours d'exécution, est codée. Le compteur ordinal contiendra alors l'adresse de l'instruction suivante. Prenons l'exemple d'un microprocesseur 8086 de INTEL :

Exemple : (**PC**)=**10000H** ; il pointe la mémoire qui contient l'instruction **MOV CX,BX** qui est codée sur deux octets (**89 D9H**) ; l'unité de commande incrémentera de deux le contenu du

PC : (**PC**) = **10002H** (la mémoire sera supposée être organisée en octets).

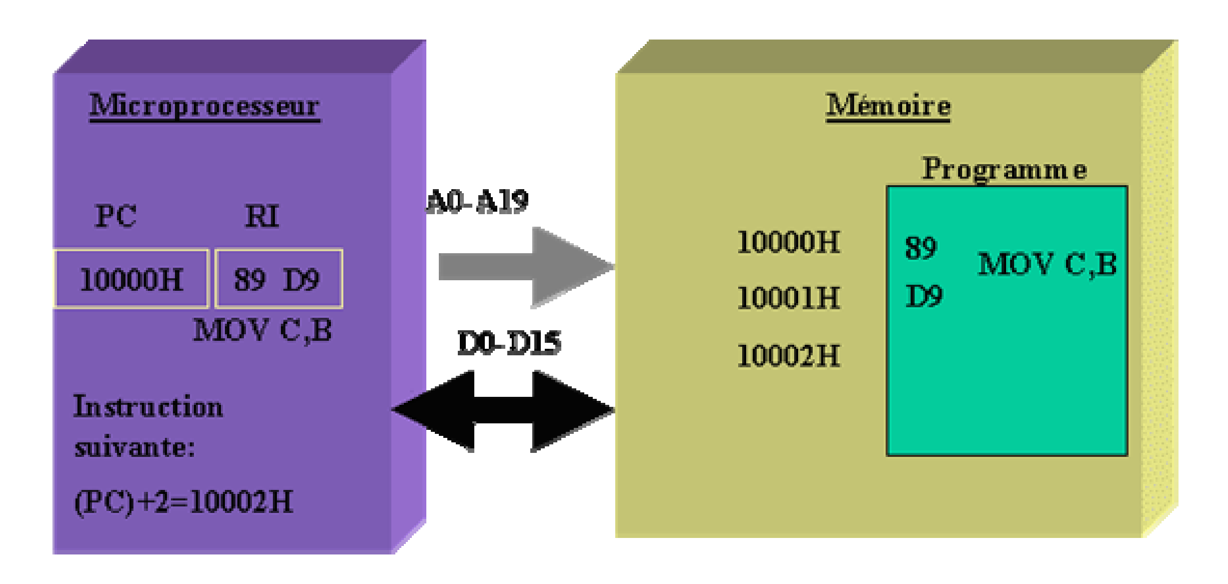


Figure 2.3 Compteur de Programme (PC).

III.1.3.2.2 Le pointeur de pile (stack pointer SP).

Il contient l'adresse de la pile. Celle-ci est une partie de la mémoire, elle permet de stocker des informations (le contenu des registres) relatives au traitement des interruptions et des sous-programmes. La pile est gérée en **LIFO** : (Last IN First Out) dernier entré premier sorti. Le fonctionnement est identique à une pile d'assiettes. Le pointeur de pile **SP** pointe le haut de la pile (**31000H** figure 2.4), il est décrémenté avant chaque empilement, et incrémenté après chaque dépilement. Il existe deux instructions pour empiler et dépiler: **PUSH** et **POP**. exemple: **PUSH A** empilera le registre A et **POP A** le dépilera. La figure suivante montre l'évolution du pointeur de pile durant l'exécution du programme commençant en **12E30H.**

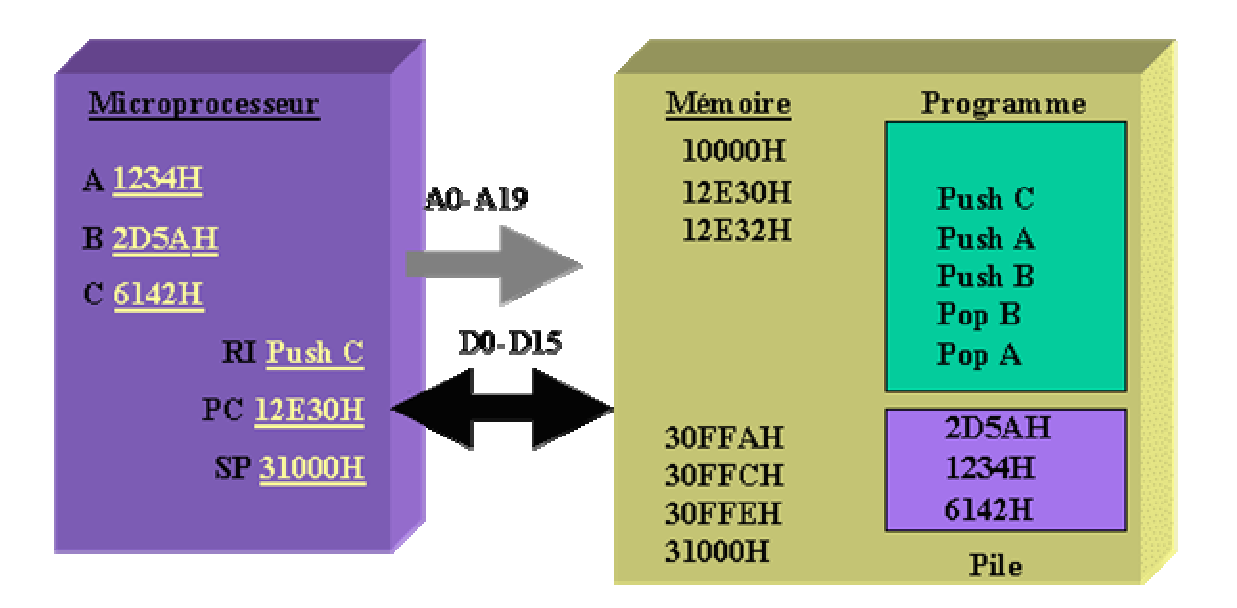


Figure 2.4. Principe de fonctionnement de la pile.

Sur cette figure le programme commence par sauvegarder le contenu de **C** dans la pile (**PUSH C**). Pour cela (**SP**) est décrémenté de deux ((**SP**)=**31000H-2=30FFEH**), puis on effectue l'écriture de (**C**) dans la mémoire à l'adresse (**SP**) : (**30FFEH**) = **6142H**. Pour **PUSH A** on obtient : (**30FFCH**)=**1234H**, et pour **PUSH B** : (**30FFAH**)=**2D5AH**. Pour l'instruction **POP B**, (**SP**) est chargé dans le registre B (**SP**=**30FFAH** ; **B**=**2D5AH**) puis (**SP**) est incrémenté de deux (**SP**= **30FFAH+2=30FFCH**). Enfin, pour **POP A** on obtient : **A**=**1234H** et (**SP**=**30FFCH + 2** = **30FFEH).**

III.1.3.2.3 Les registres d'index (index source SI et index destination DI).

Les registres d'index permettent de mémoriser une adresse particulière (par exemple : début d'un tableau). Ces registres sont aussi utilisés pour adresser la mémoire de manière différente. C'est le mode d'adressage indexé.

Exemple :

MOV A,[SI+10000H] place le contenu de la mémoire d'adresse **10000H** + le contenu de **SI**, dans le registre **A**.

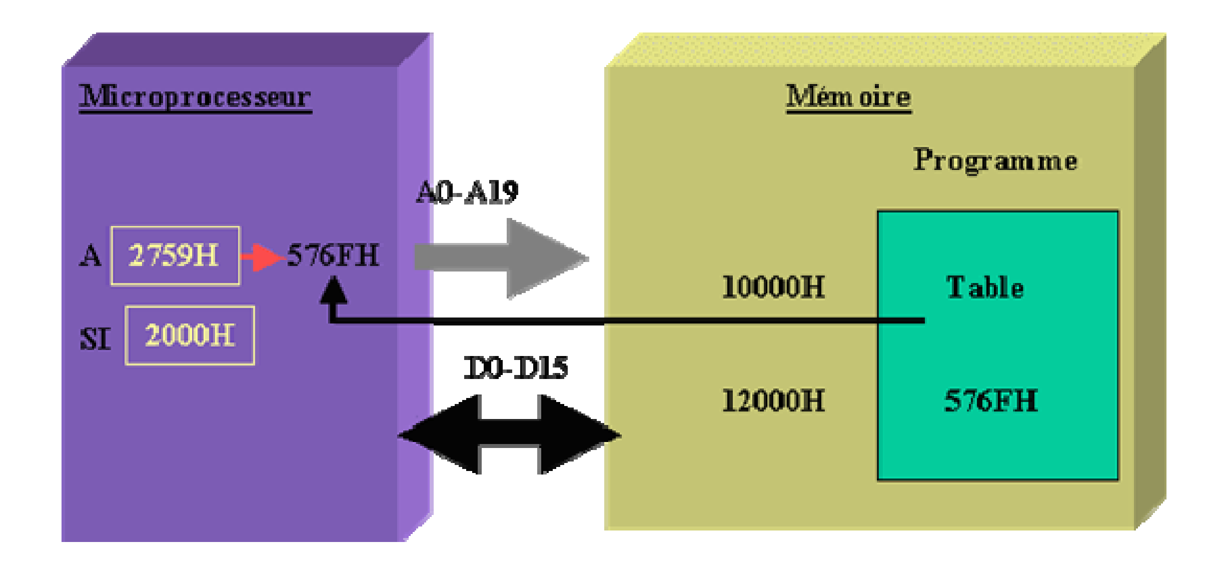


Figure 2.5. Exemple d'opération sur le registre d'index.

III.2 Principe d'exécution d'une instruction.

Dans cette architecture standard, l'exécution d'une instruction se fait en trois étapes:

-Recherche de l'instruction **(Fetch) ;**

-Décodage (**decode**) ;

-Exécution (**execute**).

III.2.1 Recherche de l'instruction.

Le contenu de PC (compteur ordinal) est placé sur le bus d'adresse (c'est l'unité de commande qui établit la connexion). L'unité de commande (UC) émet un ordre de lecture (READ=RD=1). Au bout d'un certain temps (temps d'accès à la mémoire), le contenu de la case mémoire sélectionnée est disponible sur le bus des données. L'unité de commande charge la donnée dans le registre d'instruction pour décodage. Le microprocesseur place le contenu de PC (**10000H**) sur le bus d'adresse et met RD à 1 (cycle de lecture). La mémoire met sur le bus de données le contenu de sa mémoire n° **10000H** (ici **89D9H** qui est le code de **MOV C,B**). Le microprocesseur place dans son registre d'instruction le contenu du bus de données (**89D9H**). L'unité de commande décode et exécute l'instruction **MOV C,B**.

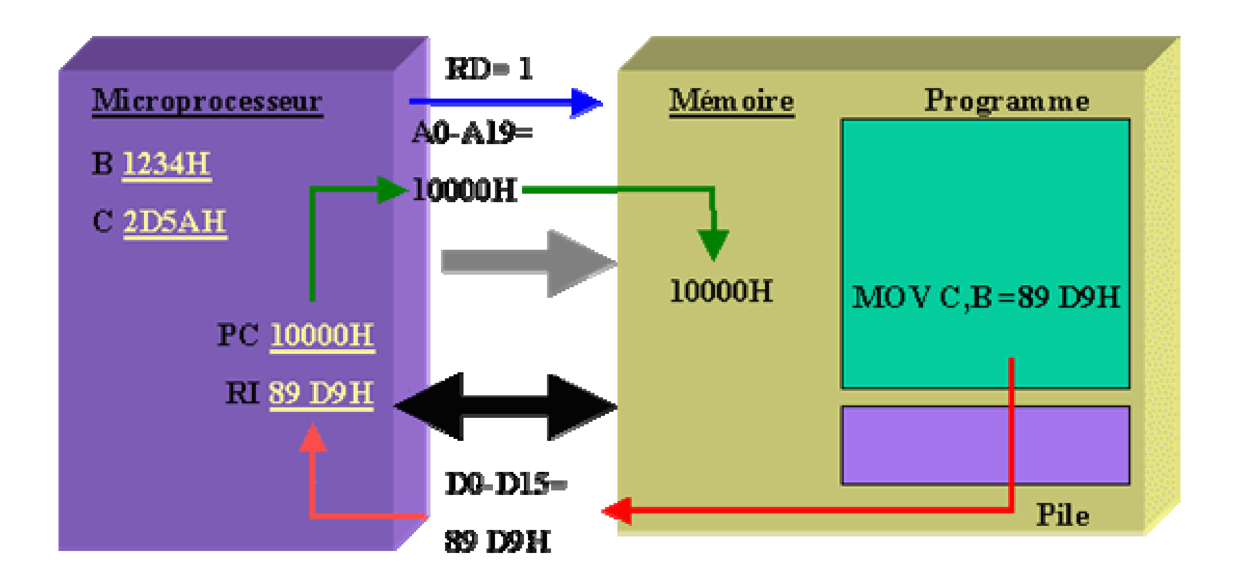


Figure 2.6 Compteur de Programme (PC).

III.2.2 Le Décodage de l'instruction.

Le registre d'instruction contient maintenant le premier mot de l'instruction qui peut être codée sur plusieurs mots. Ce premier mot contient le **code opératoire** qui définit la nature de l'opération à effectuer (addition, rotation,...) et le nombre de mots de l'instruction. L'unité de commande décode le code opératoire et peut alors exécuter l'instruction.

III.2.3 L'exécution de l'instruction.

Après l'exécution de l'instruction par micro-programme, les indicateurs sont positionnés (**O, S, Z, A, P, C**). L'unité de commande positionne le compteur ordinal (PC) pour l'instruction suivante.

IV. Les BUS.

Comme nous l'avions vu plus haut, les trois éléments fondamentaux d'un système micro-programmé sont : le microprocesseur ou microcontrôleur, la mémoire et les boîtiers d'entrées sorties. Tous ces éléments sont reliés entre eux par des bus comme le montre la figure ci dessous :

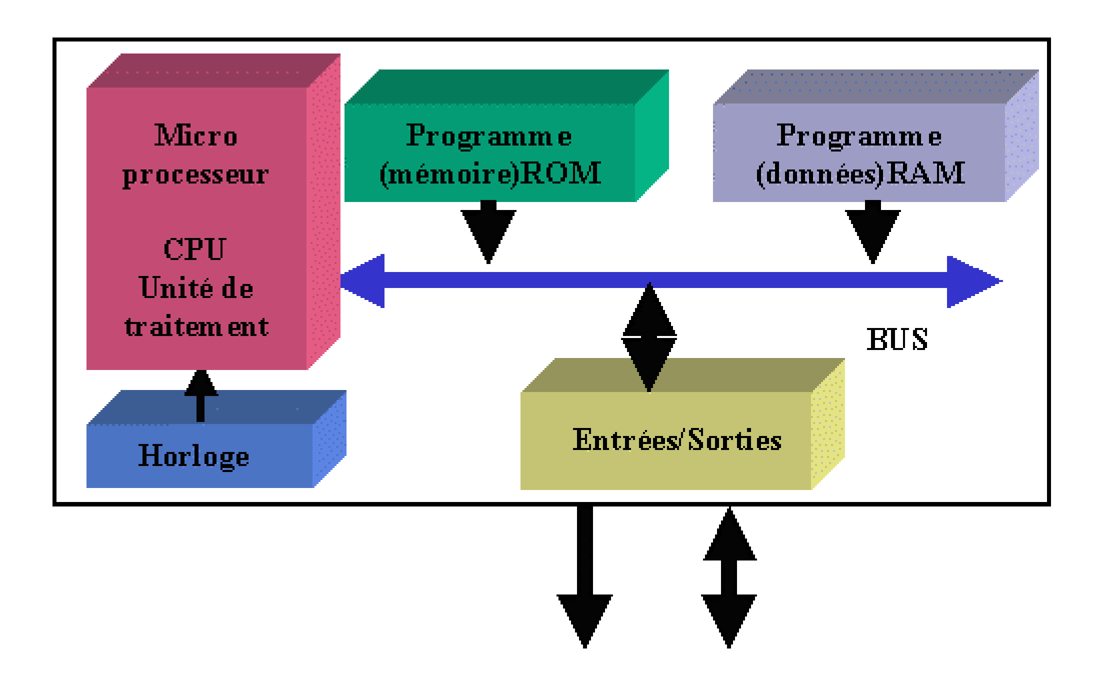


Figure 2.7. Architecture de base d'un micro-ordinateur

On appelle Bus, en informatique, un ensemble de liaisons physiques (câbles, pistes de circuits imprimés, ...) pouvant être exploitées en commun par plusieurs éléments matériels afin de communiquer. Les Bus ont pour but de réduire le nombre de traces nécessaires à la communication des différents composants en mutualisant les communications sur une seule voie de données. Dans le cas où la ligne sert uniquement à la communication de deux composants matériels, on parle parfois de port (*port série*, *port parallèle*, ...). Un Bus est caractérisé par le volume d'informations transmises simultanément (exprimé en bits), correspondant au nombre de lignes sur lesquelles les données sont envoyées de manière simultané. Une nappe de 32 fils permet ainsi de transmettre 32 bits en parallèle. On parle ainsi de "largeur de bus" pour désigner le nombre de bits qu'il peut transmettre simultanément. D'autre part, la vitesse du bus est également définie par sa fréquence (exprimée en Hertz), c'est-à-dire le nombre de paquets de données envoyés ou reçus par seconde. On parle de **cycle** pour désigner chaque envoi ou réception de données. De cette façon, il est possible de connaître la bande passante d'un bus, c'est-à-dire le débit de données qu'il peut transporter, en multipliant sa largeur par sa fréquence. Un Bus d'une largeur de 16 bits, cadencé à une fréquence de 133 Mhz possède donc une bande passante égale à :

$16 * 133.10^6 = 2128*10^6$ bit/s,

soit $2128*10^6/8 = 266*10^6$ octets/s

soit $266*10^6 /1024 = 259.7*10^3$ Ko/s

soit $259.7*10^3 /1024 = 253.7$ Mo/s

Pour la communication, un microprocesseur a besoin en général de trois Bus. Un Bus de données, un Bus d'adresse et un Bus de commande. Sur la figure suivante nous présentons un système à base de microprocesseur classique avec ces trois Bus. Chaque Bus a une fonction particulière :

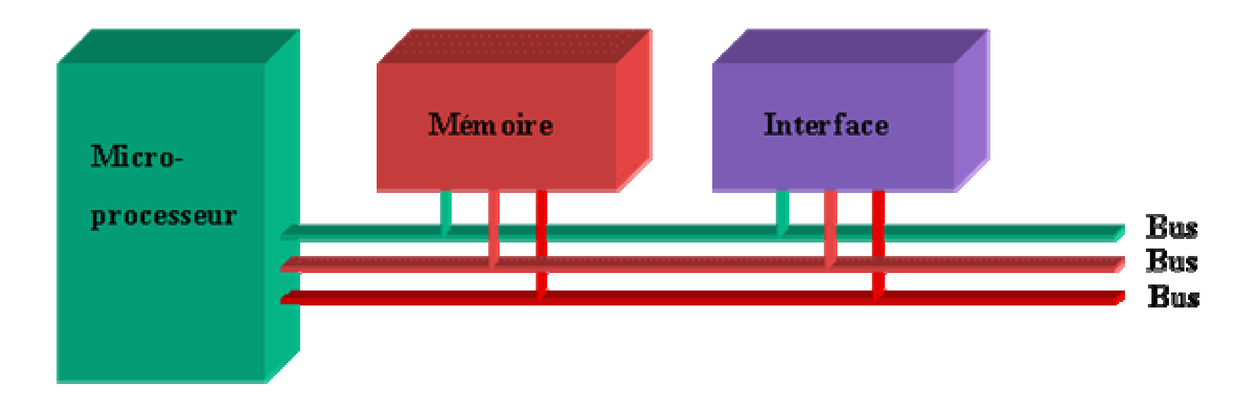


Figure 2.8. Les bus d'extensions

IV.1 Bus de données.

Il permet de véhiculer des données du microprocesseur vers un composant ou d'un composant vers le microprocesseur. Il est donc bidirectionnel. Le nombre de fils de ce bus varie suivant les microprocesseurs (8 / 16 / 32 / 64 bits). Dans la littérature, les différents fils de ce bus sont appelés D0, D1, ..., Dp-1, si le bus a "p" fils.

IV.2 Bus d'adresse.

La mémoire est composée de nombreuses cases mémoires. Chaque case est repérée par une adresse. Lorsque le microprocesseur veut, par exemple, lire une case, il doit indiquer à quelle adresse elle se trouve. Il met cette adresse sur le bus des adresses. La case mémoire reconnaît alors son adresse et met sur le bus de données son contenu.

Exemple : Bus d'adresse 16 bits - données sur 8 bits.

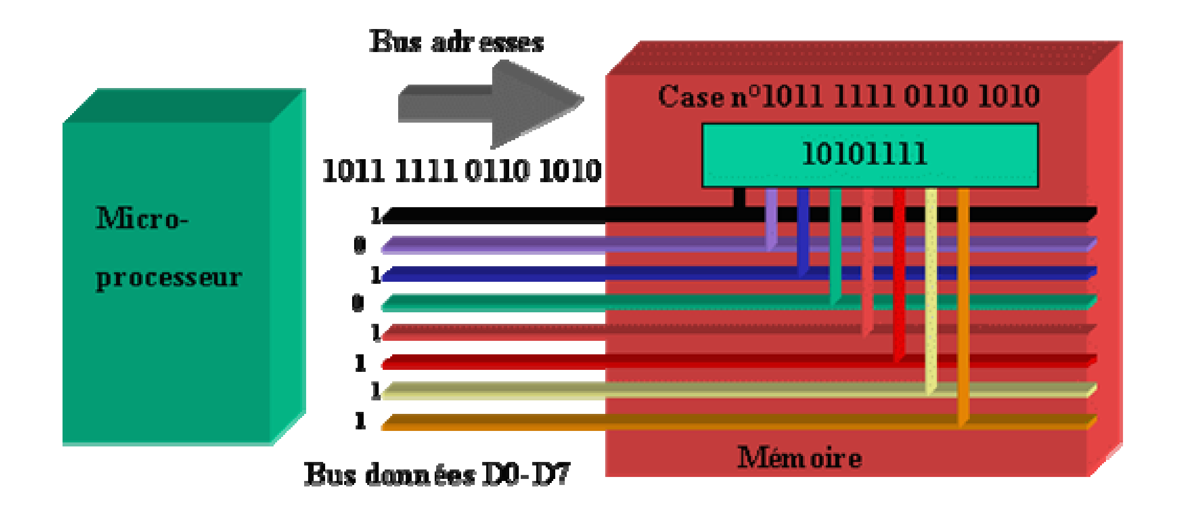


Figure 2.9. Adressage de la mémoire.

Dans l'exemple précédent, le microprocesseur écrit la donnée 10101111 dans la case mémoire d'adresse 1011 1111 0110 1010.

Le bus d'adresse est unidirectionnel : du microprocesseur vers les autres composants. Il se compose de 16 à 32 fils, suivant les microprocesseurs, que l'on nomme A0, A1, ..., An-1. Le tableau suivant donne l'espace mémoire adressable en fonction du nombre des lignes d'adresses.

16 bits	➡	adressage de 2^{16}	➡	64x1024 mots = 64 Kmots
20 bits	➡	adressage de 2^{20}	➡	1024x1024 mots = 1Mmots
32 bits	➡	adressage de 2^{32}	➡	4096x1024 x1024 mots = 4 Gmots

Tableau 1. Espace mémoire adressable en fonction des lignes d'adresses.

IV.3 Bus des commandes.

Le bus des commandes est constitué d'un ensemble de fils de "commandes", permettant la synchronisation et bien sûr la commande des boîtiers mémoires et entrées/sorties par le microprocesseur. Dans le cas précédent, la cellule mémoire doit savoir à quel instant elle doit mettre son contenu sur le bus de données. Pour cela, le microprocesseur possède une broche appelée Read ($\overline{RD}$) qu'il met à 0 (0v) lorsque la cellule doit agir. De même, lors d'une écriture du microprocesseur vers la

cellule, il met sa broche Write ($\overline{WR}$) à 0 (0V). Les signaux RD et WR sont des signaux de synchronisation, de contrôle et de commande. Ils sont reliés aux autres composants par un bus: le bus des commandes. Celui-ci comporte d'autres signaux de commandes.

V. Les Mémoires.

La mémoire vive, généralement appelée RAM (*Random Access Memory*, *mémoire à accès aléatoire*), est la mémoire principale du système, c'est-à-dire qu'il s'agit d'un espace permettant de stocker de manière temporaire des données lors de l'exécution d'un programme. En effet le stockage de données dans la mémoire vive est temporaire, contrairement au stockage de données sur une mémoire de masse telle que le disque dur, car elle permet uniquement de stocker des données tant qu'elle est alimentée électriquement. Ainsi, à chaque fois que l'ordinateur est éteint, toutes les données présentes en mémoire sont irrémédiablement effacées.

La mémoire morte, appelée ROM pour *Read Only Memory* (*mémoire à lecture seulement*) est un type de mémoire permettant de conserver les informations qui y sont contenues même lorsque la mémoire n'est plus alimentée électriquement. A la base ce type de mémoire ne peut être accédée qu'en lecture. Toutefois il est désormais possible d'enregistrer des informations dans certaines mémoires de type *ROM*.

V.1 Fonctionnement de la mémoire vive.

La mémoire vive est constituée de centaines de milliers de petits condensateurs emmagasinant des charges. Lorsqu'il est chargé, l'état logique du condensateur est égal à 1, dans le cas contraire il est à 0, ce qui signifie que chaque condensateur représente un bit de la mémoire. Etant donné que les condensateurs se déchargent, il faut constamment les recharger (le terme exact est *rafraîchir*) à un intervalle de temps régulier appelé **cycle de rafraîchissement** (d'une durée d'environ 15 nanosecondes (ns) pour une mémoire DRAM). Chaque condensateur est couplé à un transistor (de type *MOS*) permettant de "récupérer" ou de modifier l'état du condensateur. Ces transistors sont rangés sous forme de tableau (matrice), c'est-à-dire que l'on accède à une "case mémoire" (aussi appelée *point mémoire*) par une ligne et une colonne. Un boîtier mémoire est donc constitué d'un ensemble d'entités mémoire élémentaires (cellules mémoire) stockant un élément binaire (bit : **B**inary dig**IT)** ayant pour valeur 0 ou 1**.** Ces cellules sont groupées en mot **(word) de p bits (en général p=1 ou 8 bits**). Le nombre n de

cases mémoire de p bits appelé capacité ou taille de la mémoire s'exprime en Kilo **(1Ko=2^{10}=1024)** ou en Méga (1Mo=2^{20}=1024*1024=1048576).

Chaque point mémoire est donc caractérisé par une adresse, correspondant à un numéro de ligne et un numéro de colonne. Or cet accès n'est pas instantané et s'effectue pendant un délai appelé **temps de latence**. Par conséquent l'accès à une donnée en mémoire dure un temps égal au temps de cycle auquel il faut ajouter le temps de latence. Ainsi, pour une mémoire de type DRAM, le temps d'accès est **de 60 nanosecondes** (**35 ns** de délai de cycle et **25 ns** de temps

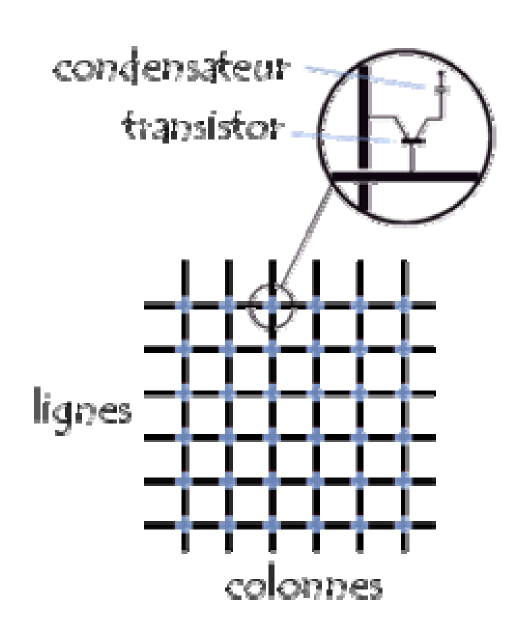


Figure 2.9. Schéma interne d'une mémoire vive.

de latence). Sur un ordinateur, le temps de cycle correspond à l'inverse de la fréquence de l'horloge, par exemple pour un ordinateur cadencé à 200 Mhz, le temps de cycle est de 5ns ($1/(200.10^6)$). Par conséquent un ordinateur ayant une fréquence élevée et utilisant des mémoires dont le temps d'accès est beaucoup plus long que le temps de cycle du processeur doit effectuer des **cycles d'attente** (en anglais ***wait state***) pour accéder à la mémoire. Dans le cas d'un ordinateur cadencé à 200Mhz utilisant des mémoires de types DRAM (dont le temps d'accès est de 60ns), il y a **11 cycles d'attente** pour un cycle de transfert. Les performances de l'ordinateur sont d'autant diminuées qu'il y a de cycles d'attentes, il est donc conseillé d'utiliser des mémoires plus rapides.

Les mémoires sont connectées à un bus d'adresse de n bits, un bus de données de p bits et des lignes de commandes (figure 2.8). Pour pouvoir communiquer avec le microprocesseur, on va

relier leurs bus ensembles. Pour cela, il est nécessaire d'avoir adéquation entre le nombre de bits des bus de données et d’adresse de la mémoire et du microprocesseur.

V.2 Sélection d’une case mémoire.

Sur la figure 2.10 apparaît une broche de validation. Elle permet de sélectionner un boîtier mémoire parmi plusieurs, d'où son appellation : "chip select". Cette broche permet d’éviter les conflits sur le bus de données. En effet dans le cas général, il existe plusieurs boîtiers mémoire sur la carte, tous branchés sur le même bus de données. Dans ce cas, il est nécessaire de construire un signal qui permettra à un seul boîtier d'accéder au bus de données. Ce signal

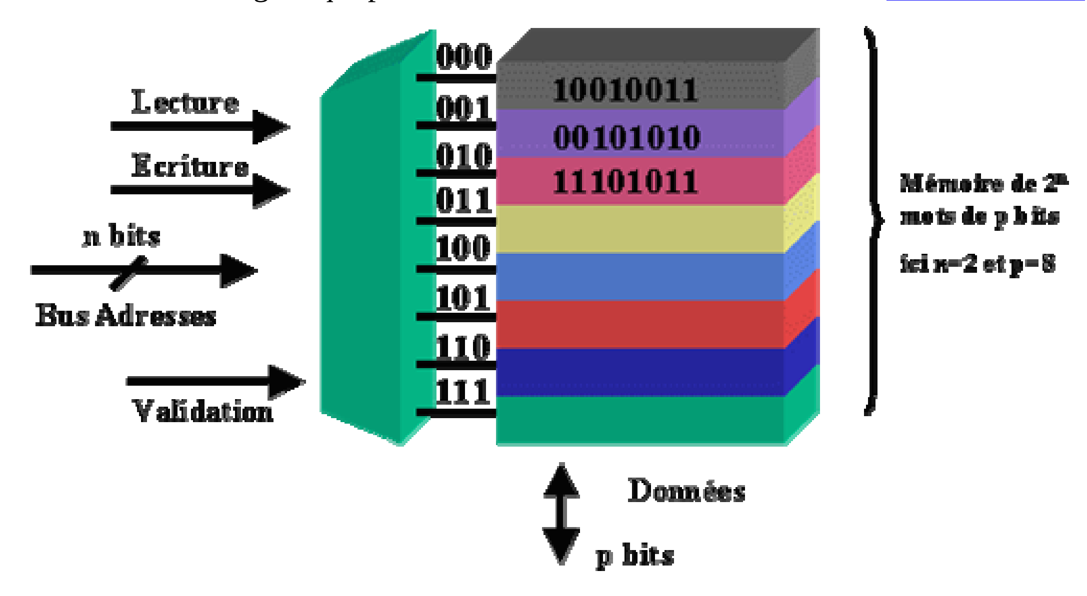


Figure 2.10. Organisation externe de la mémoire.

est appelé CS (chip select) sélection de boîtier ou CE (chip enable) validation de boîtier. Il faut créer autant de CS qu'il y a de boîtiers. Dans notre exemple figure 2.11, il nous faut fabriquer deux CS : CS1, CS2

Exemple :

Le bus d’adresse est sur 16 bits, le bus de données est sur 8 bits (figure 2.11). A l'adresse 1000H, le premier boîtier mémoire contient A7H et le second contient A6H. Si le microprocesseur fait une lecture à l'adresse 1000H (RD=1). Le premier boîtier mettra **A7**H sur le bus de données et le deuxième **A6H** sur le bus de données. Le bus de données D7- D0 a donc sur son fil D0 un **"0**" et un "**1**" ; c'est à dire 5 volts et la masse. C'est donc un **court-circuit.**

Les sélections de boîtiers "CS" (CS1 et CS2 sur la figure suivante) sont des fonctions logiques. Elles proviennent de circuits combinatoires appelés "logique de décodage ou encore décodage adresse". Les variables logiques de ces fonctions logiques sont

les variables du bus d'adresse (A0-An-1). Le choix des plages de validation des CSi sont exclusives les unes par rapport aux autres. C'est à dire qu'elles ne se recouvrent pas.

Si CSi(A0-An-1) = 1 =>CSj(A0-An-1) = 0 quelque soit j ≠ i

Exemple:

Prenons l'exemple de la figure 2.13, si le bus d'adresse se compose de 16 fils (A0-A15). Supposons que la taille mémoire des deux boîtiers soit 32*1024 = 32K = 2^{15} adresses.

Il y a donc 15 broches adresses sur chaque boîtier. Nous pourrons mettre les fils A0-A14 du bus d'adresse sur ces broches. On peut prendre CS1 = /A15 (complément de A15) et

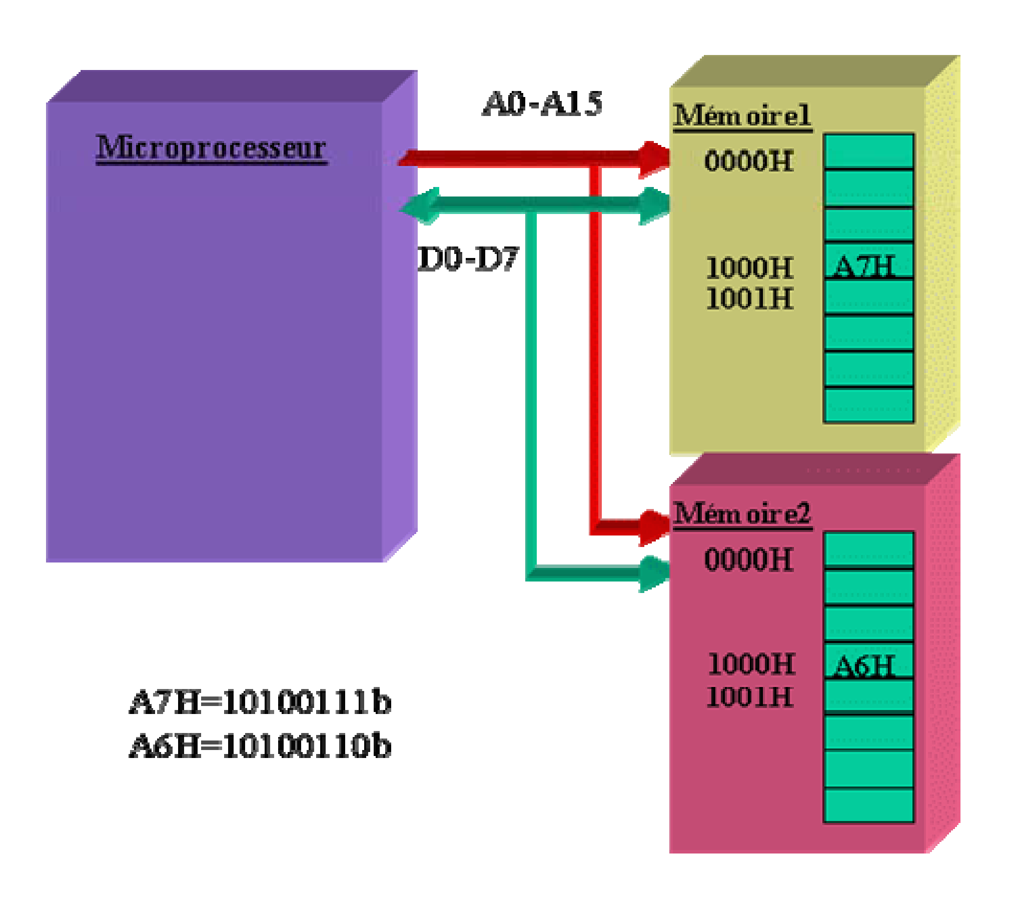


Figure 2.11 Conflit sur un bus.

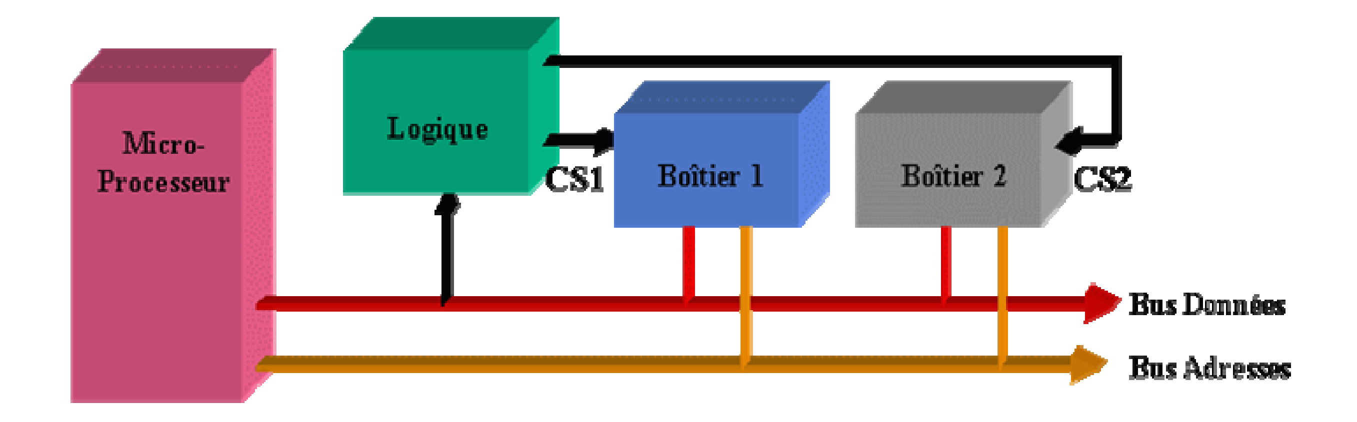


Figure 2.12 Sélection de la mémoire.

CS2=A15. Une lecture du microprocesseur à l'adresse **1000H** donnera : CS1 =1 et CS2 =0. Ce sera donc le premier boîtier qui sera validé et qui mettra le contenu de sa mémoire n° 0001 0000 0000 0000 sur le bus de données **(A7H** si on reprend l'exercice précédent). L'assemblage de plusieurs boîtiers forme un plan mémoire de plus grande capacité. L'assemblage horizontal (en largeur) permet de réaliser des mémoires de mots plus grands. Les boîtiers partagent le même bus d'adresses et de contrôle. Tandis que l'assemblage vertical (en profondeur) augmente la capacité (taille) mémoire du micro-ordinateur, les boîtiers partagent le même bus de données.

Exemple :

Nous disposons d'un microprocesseur utilisant un bus de données de dimension 16 fils (D0-D15) et pouvant adresser 1 Mo cases de mémoire (bus d'adresse sur 20 fils A0-A19 : 2^{20}=1Mo). Nous disposons également de boîtiers mémoire de 128 Ko octets chacun. De plus, nous désirons travailler sur des mots de 16 bits. Chaque boîtier mémoire a besoin de17 fils du bus d'adresse A1-A17 (2^{17}=128K) pour être branchés sur ses broches d'adresse. Il reste donc A0, A18 et A19. On va se servir de A18 et A19 pour construire les quatre CS des 8 boîtiers mémoire et de A0 pour sélectionner partie basse ou haute du bus de données. La solution de ce problème se trouve dans la figure (2.15), où nous utilisons un décodeur d'adresse à deux entrées de sélection et quatre sorties (CSi). On a immédiatement l'expression des fonctions logiques CS0, CS1, CS2, CS3. (CS0=/A19*/A18 CS1=/A19*A18 CS2=A19*/A18 CS3=A19*A18)

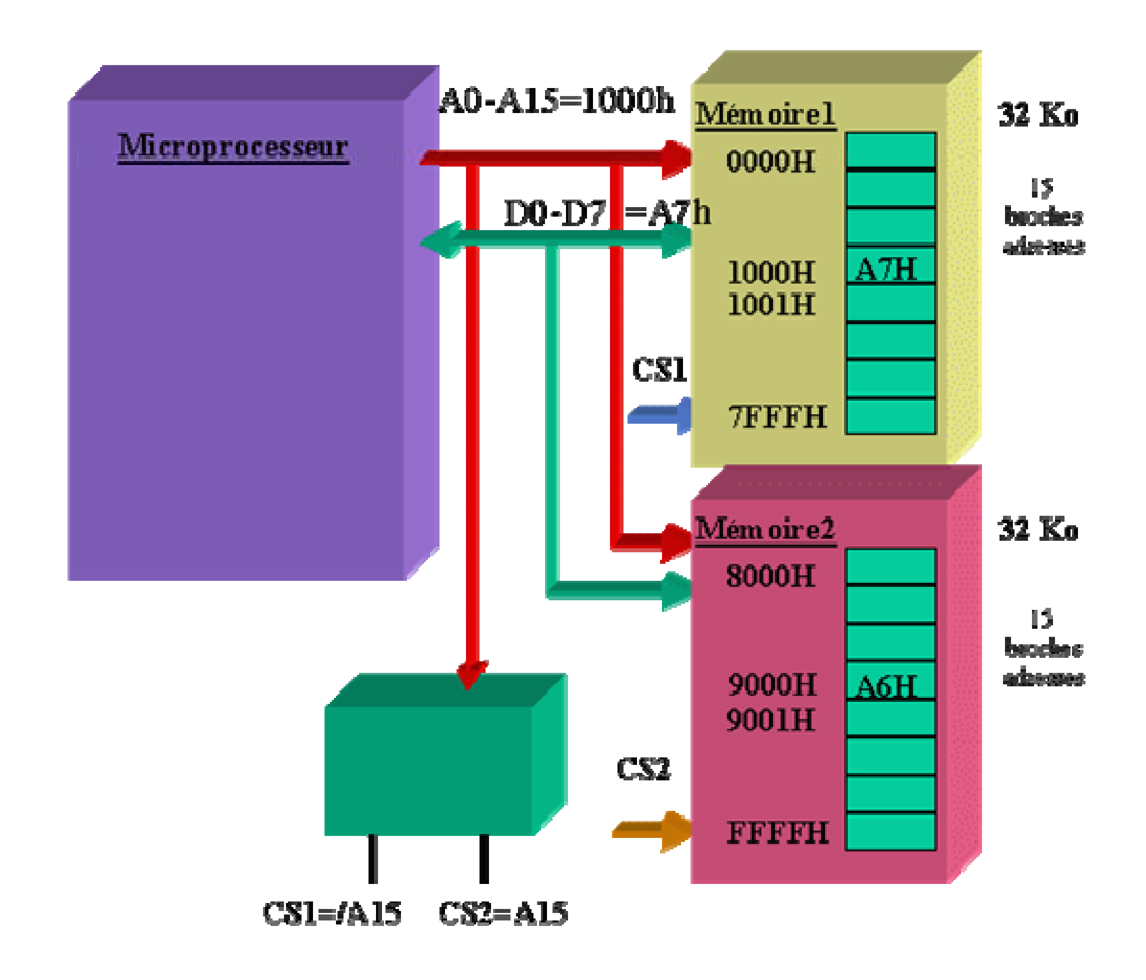
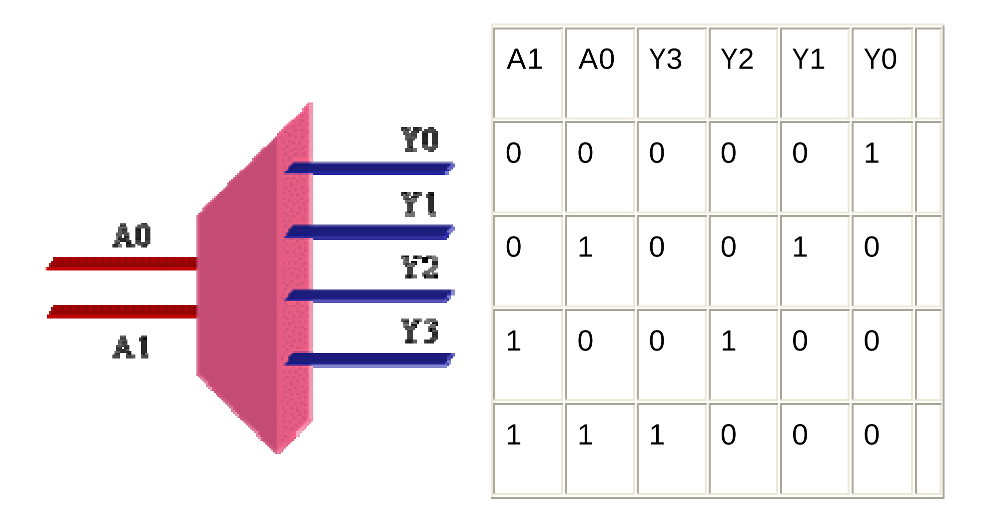


Figure 2.13 Réalisation d'une sélection boîtier.

V.3 Rappel sur les décodeurs.



A1	A0	Y3	Y2	Y1	Y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Figure 2.14. Le décodeur 2 -> 4.

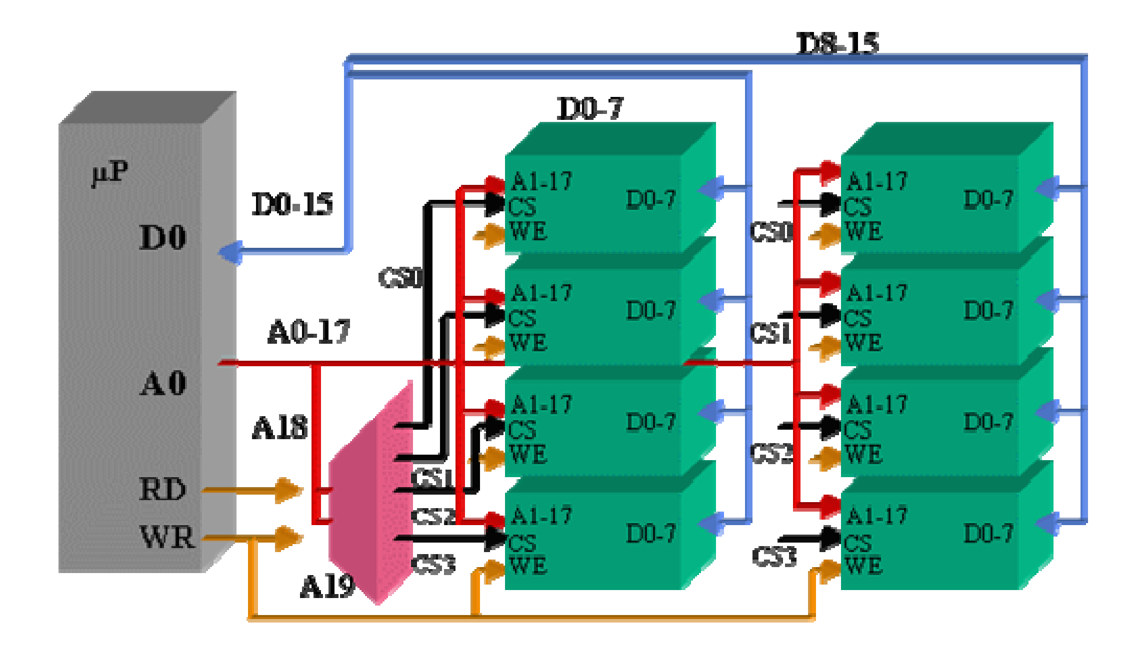


Figure 2.15. Réalisation du plan mémoire.

Le décodeur positionne à 1 (5v) la sortie n° "A1A0". Exemple: A1A0=11b (binaire) = 3d (décimal) =>Y3Y2Y1Y0=(1000)b. Dans le cas de la figure ci dessus les entrées A1 et A0 du décodeur sont les lignes d'adresse A19 et A18 du microprocesseur, ce qui permet de sélectionner 128 Ko par sortie Yi du décodeur (CS1, CS2, CS3 et CS4). La mémoire totale sélectionnée est alors de 512 Ko par banc mémoire. Soit au total 1Mo de cases mémoires sélectionnées.

I. Introduction.

Dans cette partie du cours, nous allons étudier la programmation en langage assembleur d'un microprocesseur de la famille INTEL. L'étude complète d'un processeur réel, comme le 80486 ou le Pentium fabriqués par Intel, dépasse largement le cadre de ce cours : le nombre d'instructions et de registres est très élevé. Nous allons ici nous limiter à un sous-ensemble du microprocesseur 80486 (seuls les registres et les instructions les plus simples seront étudiés). De cette façon, nous pourrons tester sur un PC les programmes en langage machine que nous écrivons. La gamme de microprocesseurs 80x86 équipe les micro-ordinateurs de type PC et compatibles. Les premiers modèles de PC, commercialisés au début des années 1980, utilisaient le 8086, un microprocesseur 16 bits. Les modèles suivants ont utilisé successivement le 80286, 80386, 80486 et Pentium (ou 80586). Chacun de ces processeurs est plus puissant que les précédents : augmentation de la fréquence d'horloge, de la largeur de bus (32 bits d'adresse et de données), introduction de nouvelles instructions (par exemple calcul sur les réels) et ajout de registres. Chacun d'entre eux est *compatible* avec les modèles précédents; un programme écrit dans le langage machine du 80286 peut s'exécuter sans modification sur un 486. L'inverse n'est pas vrai, puisque chaque génération a ajouté des instructions nouvelles. On parle donc de *compatibilité ascendante*. Du fait de cette compatibilité, il est possible de programmer le Pentium comme un processeur 16 bits. C'est ce que nous ferons dans nos exemples par souci de simplification. Ainsi, nous n'utiliserons que des registres de 16 bits.

II. Architecture interne du microprocesseur 8086.

Quoique, nous ayons déjà parlé de l'architecture de base d'un microprocesseur, nous pensons qu'il est utile d'en reparler. Nous allons nous intéresser ici à l'architecture interne du microprocesseur 8086, sujet de cette étude, avant d'entamer l'étude des instructions de ce même microprocesseur. Du fait de la compatibilité, ascendante des microprocesseurs de INTEL, il est possible de programmer le Pentium avec les instructions du 8086 que nous allons étudier dans cette partie. Les registres du microprocesseur 8086 sont tous des registres 16 bits. Ils sont répartis en 5 catégories: registres de travail, registres de segment, registres pointeurs, registres index et registres spéciaux. Les **registres de travail** sont au nombre de 4 et sont notés ***AX, BX, CX, DX***. Chacun d'eux peut être utilisé comme un registre **16 bits** ou décomposé en 2 registres 8 bits au gré du programmeur. Le registre ***AX*** peut être décomposé en ***AH*** (H pour

High, les 8 bits de fort poids de *AX*) et ***AL*** (L pour Low, les 8 bits de faible poids de *AX*). De la même façon ***BX*** peut être décomposé en ***BH*** et ***BL***, ***CX*** en ***CH*** et ***CL***, ***DX*** en ***DH*** **et** ***DL***. Dans beaucoup d'instructions, ces registres sont banalisés et peuvent être utilisés indifféremment. Toutefois, dans certaines instructions, ils ont un rôle bien précis: le registre *AX* est utilisé implicitement comme accumulateur (dans les instructions de multiplication), *BX* comme registre de base (pour des données dans le segment pointé par le registre de segment ***ES***), *CX* comme compteur d'itérations (dans les instructions de boucle) et ***DX*** contient l'adresse du port **d'E/S** dans les instructions d'E/S *IN* et *OUT*.

Les **registres de segment** sont aussi au nombre de 4 et sont notés ***CS*, *DS*, *SS* et *ES***. Pendant l'exécution d'un programme, ils contiennent les numéros des segments accessibles. Le registre *CS* contient le numéro du segment de code (Code Segment), *DS* le numéro du segment de données (Data Segment), *SS* le numéro du segment de pile (Stack Segment) et *ES* le numéro du segment de données supplémentaire (Extra Segment). L'utilisation de ces registres est implicite sauf indication explicite du programmeur. Les **registres pointeurs** sont au nombre de 2 et notés ***SP* et *BP***. Ils sont dédiés à l'utilisation de la pile. Le registre *SP* (Stack Pointer) pointe sur le sommet de la pile et il est mis à jour automatiquement par les instructions d'empilement et de dépilement; *BP* (Base Pointer) pointe la base de la région de la pile contenant les données accessibles (variables locales, paramètres,...) à l'intérieur d'une procédure. Il doit être mis à jour par le programmeur. Les **registres d'index** sont au nombre de 2 et sont notés ***SI* et *DI***. Chacun de ces 2 registres peut être utilisé pour indexer les éléments d'un tableau. Dans les instructions de mouvement de chaînes d'octets, ils sont utilisés simultanément: *SI* pour indexer les caractères de la chaîne émettrice, d'où son nom Source Index et *DI* pour les caractères de la chaîne réceptrice, d'où son nom *Destination Index*. Les **registres spéciaux** sont au nombre de 2 et notés ***IP*** (Instruction Pointer) et ***SR*** (Status Register). *IP* joue le rôle de compteur ordinal et contient le déplacement dans le segment de code de la prochaine instruction à exécuter. Le couple *CS:IP* donne donc l'adresse physique sur 20 bits. Le registre *SR* contient l'état du microprocesseur matérialisé par les indicateurs (Flags): *O* pour Overflow, *D* pour Direction, *I* pour Interrupt, *T* pour Trace, *S* pour Sign, *Z* pour Zero, *A* pour Auxiliary carry, *P* pour Parity et *C* pour Carry.

				O	**D**	**I**	**T**	**S**	**Z**		**A**		**P**		**C**

Figure 3.1 Registre d'état SR

III. Organisation de l'espace adressable.

L'espace adressé par le 8086 est un espace de 1 Moctets organisé en **segments** ayant des tailles pouvant aller jusqu'à 64K octets. Ces segments peuvent se chevaucher et leurs adresses début sont multiples de 16: ainsi les segments de numéros 0, 1, 2, 3, 4, 5 ... auront les adresses physiques 0, 16, 32, 48, 64, 80...Les adresses sont des adresses segmentées: formées par le couple (numéro de segment, déplacement). Le numéro de segment et le déplacement sont codés chacun sur 16 bits. Cette structuration répond au découpage logique des programmes. Elle offre en outre l'avantage de ne spécifier que le déplacement dans les instructions à référence mémoire, le numéro de segment étant implicite (il demeure le même tant qu'on n'accède qu'à des données ou instructions situées dans un même segment). Ainsi la plupart de ces instructions contiennent une adresse de 16 bits au lieu de 20 bits. L'adresse physique envoyée par le 8086 sur le bus est une adresse 20 bits. Elle est calculée à partir du numéro de segment (conservé dans un registre) et du déplacement fourni par l'instruction: elle est obtenue en additionnant le numéro de segment décalé de 4 positions vers la gauche (ce qui donne l'adresse physique du début du segment sur 20 bits) au déplacement sur 16 bits.

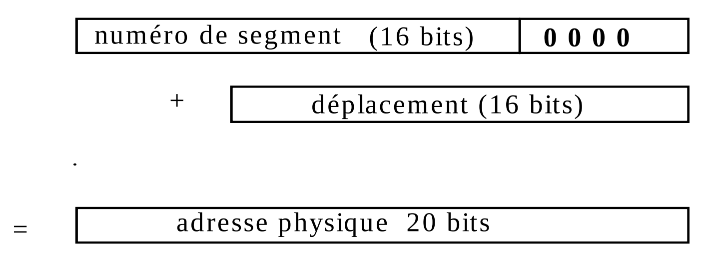


Figure 3.2. Formation d'une adresse physique par le microprocesseur

IV. Modes d'adressage.

Les instructions peuvent avoir 0, 1, ou 2 opérandes.

Exemples :

- Instructions sans opérande: *NOP*, *STI*, *CLI*, *PUSHF*, *CBW*...;

- Instructions avec une seule opérande: *INC*, *DEC*, *NEG*, *NOT* ....;

- Instructions avec deux opérandes: *CMP*, *ADD*, *MOV*, *LEA*, *XCHG*, *AND* ......

Une opérande peut se trouver dans un registre du 8086, dans un octet, dans un mot 16 bits ou dans un double mot 32 bits.

Les opérations peuvent avoir lieu:

- entre un registre et un autre registre,

- entre un registre et un octet ou un mot en mémoire *mais pas entre 2 octets ou 2 mots en mémoire* (il faut passer dans ce cas par un registre). Toutefois les instructions de mouvement de chaînes d'octets effectuent "directement" le transfert de mémoire à mémoire (octet par octet ou mot par mot), l'adressage se faisant par les registres DS:SI et ES:DI.

IV.1 adressage registre à registre.

Les 2 opérandes se trouvent dans des registres.

Exemple: *MOV AX*, *BX* ; opérandes 16 bits

ADD CH, *DL* ; opérandes 8 bits

IV.2 adressage immédiat .

L'une des opérandes est une constante codée dans l'instruction elle même.

Exemples: *MOV AX*, 0A1EBH ; *AX*:= valeur hexa A1EB

ADD CL, 25 ; *CL*:= *CL* + 25

OR AH,11000000 ; forcer les 2 bits de fort poids de *AH* à 1

IV.3 adressage direct.

Le déplacement de l'un des opérandes est spécifié dans l'instruction.

Exemples: *MOV BX*,*Total*

MOV DX,*ES*:*Nom*

Dans cet exemple *Total* est l'identificateur d'une variable 16 bits dans le segment de données basé par *DS*. Le symbole *Total* sera remplacé dans l'instruction par la valeur

numérique du déplacement de la variable par rapport à ce segment. Par contre la variable désignée par *Nom* se trouve dans le segment de données basé par *ES*.

IV.4 adressage indirect (ou basé).

Un registre est utilisé pour contenir l'adresse d'une opérande; ce registre peut être *BX* ou *BP*.

Exemples: *MOV AX*,[*BX*]

MOV AX,[*BP*]

Dans cet exemple *BX* contient l'adresse de l'opérande dans le segment de données *DS (*adresse physique:= (*DS*)*16+(*BX*)). *BP* contient l'adresse de l'opérande dans le segment *SS (*adresse physique:= (*SS*)*16+(*BP*)). L'avantage de ce mode d'adressage est qu'il permet avec la même instruction exécutée plusieurs fois (dans une boucle par exemple) d'accéder à des variables différentes en modifiant entre 2 exécutions le contenu du registre *BX* ou *BP*.

IV.5 adressage indexé.

Ici, on utilise un index qui est mis dans le registre *SI* ou *DI*.

Exemple: *MOV AX*,*Tab*[*SI*]

Dans cet exemple *SI* contient l'index dans le tableau *Tab* de l'élément référencé. L'adresse physique est égale à:= (*DS*)*16+ déplacement de *Tab* + (*SI*).

IV.6 adressage indirect indexé (ou basé indexé).

Ici, on utilise un registre d'indirection (*BX* ou *BP*) et un registre d'index (*SI* ou *DI*).

Exemples: *MOV AX*,[*BX*][*SI*]

MOV AX,[*BP*][*SI*]

Dans cet exemple, dans la première instruction, l'adresse de la table est donnée par *BX* et l'index dans la table par *SI :* adresse physique:= (*DS*)*16+(*BX*)+(*SI*). Cette instruction peut accéder aux éléments de 2 tables différentes en changeant entre 2 exécutions le contenu de *BX*. Par contre dans la deuxième instruction, l'adresse de la table se trouve dans la pile, son déplacement est dans *BP :* adresse physique:= (*SS*)*16+(*BP*)+(*SI*).

IV.7 adressage basé indexé avec déplacement.

Ce mode d'adressage est utile pour accéder à un élément d'une structure (enregistrement).

Exemple: *MOV AX,Compte[BX][SI]*

Dans cet exemple, *Compte* est le déplacement de l'enregistrement dans le segment de données, *BX* pointe un champ de l'enregistrement et *SI* un élément dans le champ: adresse physique:= (*DS*)*16+*Compte*+(*BX*)+(*SI*).

V. les principales instructions de l'assembleur X86.

V.1 Les instructions de transfert de données.

V.1.1 L'instruction MOV.

Syntaxe: *MOV Destination,Source*

Destination = registre/case mémoire.

Source = registre/case mémoire/valeur immédiate.

Exemple: *MOV AX,CX :* charger dans *AX* le contenu de *CX.* Le contenu de *CX* reste inchangé.

V.1.2 L'instruction LEA (Load Effective Address).

Syntaxe: *LEA registre,opérande*

registre = registre 16 bits

opérande = nom d'étiquette (déplacement 16 bits).

L'instruction *LEA* peut être remplacée par l'instruction *MOV* contenant la directive *OFFSET*.

Exemples: *LEA BX,Tableau* ou bien *MOV BX,OFFSET Tableau*

LEA BX,Tableau[SI]

Dans la première instruction, nous chargeons l'adresse de *Tableau* dans *BX*, par contre dans la deuxième instruction nous chargeons l'adresse contenu dans *Tableau* indéxé avec *SI* dans *BX*.

V.1.3 L'instruction XCHG.

Syntaxe: *XCHG opérande1*,opérande2

opérande1 = registre de donnée/case mémoire

opérande2 = registre de donnée

Exemple: *XCHG AX*,*Somme*: échange des contenus de *AX* et de *Somme*

V.1.4 Les instructions PUSH et POP.

Syntaxe: *PUSH Source*

POP Destination

Source = registre ou case mémoire dont le contenu doit être placé sur la pile

Destination = registre ou case mémoire auquel est affectée la valeur retirée du sommet de la pile, les éléments de la pile étant des mots 16 bits. *Source* et *Destination* doivent être sur 16 bits.

Exemple: *PUSH SI* empile le contenu de *SI*

Remarque: Pour empiler/dépiler le registre d'état *SR*, il faut utiliser les instructions *PUSHF* et *POPF* (F pour Flags).

V.2 Les instructions arithmétiques.

V.2.1 Les instructions d'addition ADD et ADC.

Syntaxe: *ADD Destination*,*Source*

ADC Destination,*Source*

Destination = registre/case mémoire.

Source = valeur immédiate/ registre/case mémoire à condition que *Destination* ne soit pas une case mémoire.

Fonction: *ADD* fait *Destination*:= *Destination* + *Source*

ADC fait *Destination*:= *Destination* + *Source* + *Carry*

Exemple:

ADD *AX*,*BX* ;*AX* reçoit *AX* + *BX*

ADD *CX*,250 ;*CX* reçoit *CX* + 250

ADD *Cumul*,*AX* ;*Cumul* reçoit *Cumul* + *AX*

ADD *AX*,*Tab*[*SI*] ;*AX* reçoit *AX* + *Tab*[*SI*]

ADD *AX*,0F00H ;ces 2 instructions additionnent la

ADC *DX*,0 ;valeur immédiate 0F00H à la paire de registres *DX:AX (32 bits)*.

V.2.2 Les instructions de soustraction SUB et SBB.

Syntaxe: *SUB Destination*,*Source*

SBB Destination,*Source*

Destination = registre/case mémoire

Source = valeur immédiate/ registre/case mémoire à condition que *Destination* ne soit pas une case mémoire.

Fonction: *SUB* fait *Destination*:= *Destination* - *Source*

SBB fait *Destination*:= *Destination* - *Source* – *Borrow*

Borrow prend la valeur de *Carry (retenue)*.

Exemple: *SUB AX*,*DX* ;*AX* reçoit *AX-DX*

SBB SI,100 ;*SI* reçoit *SI*-100 - *Carry*

IV.2.3 Les instructions d'incrémentation et décrémentation: INC et DEC.

Syntaxe: *INC Opérande*

DEC Opérande

Opérande = registre/case mémoire

Fonction: *INC* fait +1 sur l'opérande

DEC fait -1 sur l'opérande

Exemples:

INC AX ; *AX*:= *AX* + 1

DEC CX ; *CX*:= *CX* - 1

DEC WORD PTR Valeur ; *Valeur*:= *Valeur* - 1

V.2.4 Les instructions de multiplication: MUL et IMUL.

Syntaxe: *MUL Opérande*

IMUL Opérande

Opérande = registre ou case mémoire

La deuxième opérande est implicitement dans *AL* ou *AX*.

Le résultat sera dans *AX* pour les multiplications 8 bits x 8bits. Il sera dans la paire *DX*:*AX* pour les multiplications 16 bits x 16 bits.

Exemples: *MUL CL* ; *AX*:=*AL* * *CL* résultat sur 16 bits

MUL CX ; *DX*:*AX*:= *AX* * *CX*; résultat sur 32 bits

MUL ignore le signe des opérandes.

IMUL tient compte du signe des opérandes.

V.2.5 Les instructions de division: DIV et IDIV.

Syntaxe: *DIV Opérande*

IDIV Opérande

Opérande = registre / case mémoire

Le dividende est dans *AX* ou dans la paire *DX*:*AX*.

Fonction:

- division d'opérandes 8 bits: division du contenu de *AX* par l'opérande (8 bits), le quotient est dans *AL*, le reste dans *AH*

- division d'opérandes 16 bits: division du contenu de *DX*:*AX* par l'opérande 16 bits, le quotient est dans *AX*, le reste dans *DX*.

DIV ignore le signe des opérandes.

IDIV tient compte du signe des opérandes.

Exemples:

DIV BX ;*DX*:*AX* est divisé par *BX*, le reste est dans *DX* et le quotient dans *AX*

DIV BL ;*AX* est divisé par *BL*, le reste est dans *AH*, le quotient dans *AL*.

V.3 Les instructions logiques AND, OR et XOR.

Syntaxe: *AND Destination*,*Source*

OR Destination,*Source*

XOR Destination,*Source*

Destination = registre/case mémoire

Source = registre/valeur immédiate/case mémoire à condition que *Destination* ne soit pas une case mémoire.

Exemples:

AND AX,0FFF0H ;met à 0 les 4 bits de faible poids de *AX*

OR AL,1 ;met à 1 le bit de faible poids de *AL*

AND AX,*Masque* ;fait un ET logique entre le contenu de *AX* et celui de *Masque*, résultat ;dans *AX*

XOR AX,-1 ;fait le complément à 1 de *AX*

V.4 Les instructions de rotation RCL, RCR, ROL, ROR.

Syntaxe: *RCL Opérande*,*Nombre*

RCR Opérande,*Nombre*

ROL Opérande,*Nombre*

ROR Opérande,*Nombre*

Opérande = registre ou case mémoire

Nombre = soit la valeur immédiate, soit la valeur positive contenue dans *CL*.

Fonctions et exemples: *ROL* fait la rotation à gauche de l'opérande: Si *AL* = 00110111 alors après l'instruction *ROL AL*,1 *AL* contiendra la valeur 01101110. *ROR* fait la rotation à droite de l'opérande: Si *AL* = 00110111 alors après l'instruction *ROR AL*,1 *AL* contiendra la valeur 10011011. *RCL* fait la rotation à gauche de l'opérande en faisant intervenir le bit *Carry* dans la rotation (la valeur de *Carry* est introduite dans le bit *LSB* de l'opérande et *Carry* prend la valeur du bit *MSB* de l'opérande). Si par exemple *AL* contient la valeur 37h correspondant à 00110111, alors après l'instruction *RCL AL*,1 AL contiendra la valeur 01101110 si *Carry*=0 ou la valeur 01101111 si *Carry*=1. *RCR* fait la rotation à droite de l'opérande avec le bit *Carry*: Si par exemple *AL* contient la valeur binaire 00110111; alors, après l'instruction: *RCR AL,1* *AL* contiendra la valeur 10011011 si *Carry*=1 ou la valeur 00011011 si *Carry*=0.

V.5 Les instructions de comparaison: CMP et TEST.

Syntaxe: *CMP Destination*,*Source*

TEST Destination,*Source*

Destination = registre / case mémoire

Source = registre / case mémoire / valeur

Exemples:

CMP AL,*CL* ; compare *AL* à *CL*

CMP AL,0 ; compare *AL* à 0

TEST AL,*AH* ; ET logique entre *AL* et *AH*

Remarque: *TEST* laisse les opérandes du ET logique inchangées; les indicateurs sont positionnés en fonction du résultat du ET logique.

V.6 Les instructions de décalage: SAL/SAR et SHL/SHR.

Fonction: *SAL* Décalage arithmétique à gauche

SAR Décalage arithmétique à droite

SHL Décalage logique à gauche

SHR Décalage logique à droite

Syntaxe: *SAL Opérande,Nombre*

SAR Opérande,Nombre

SHL Opérande,Nombre

SHR Opérande,Nombre

Opérande = registre / case mémoire

Nombre = soit la valeur 1, soit la valeur contenue dans *CL*.

Exemples: Si *AL* contient la valeur B7H ou 10110111 binaire, alors après l'instruction *SAL AL,1 AL* contiendra la valeur 01101110. Après les instructions *MOV CL,3* et *SAL AL,CL AL* contiendra la valeur initiale 10110111 décalée de 3 positions vers la gauche c'est à dire la valeur binaire 10111000. *SAR* fait un décalage à droite de l'opérande considérée comme une quantité signée; c'est ainsi que le bit de poids fort remplace les positions décalées vers la droite. Par exemple si *AL*=10011101 et *CL*=3 alors après l'instruction *SAR AL,CL*; *AL* contiendra la valeur 11110011. Si *AL* contenait la valeur 01110111 la même instruction donnerait la valeur 00001110 dans *AL*. *SHR* et *SHL* fonctionnent comme *SAL* et *SAR* sans extension du signe.

V.7 Les instructions de saut (ou de branchement).

Les instructions de saut (appelées aussi instructions de branchement) servent à faire passer l'exécution d'un programme d'un emplacement du programme à un autre. Il y a plusieurs sortes d'instructions de saut et 3 distances de saut différentes. Un saut peut être :

- *court (SHORT), rapproché (NEAR) ou éloigné (FAR).*

- direct ou indirect.

- inconditionnel ou conditionnel.

Petit saut (*SHORT jump*) : Une destination de saut sera de ce type si le saut ne dépasse pas 127 octets en avant ou 128 octets en arrière (par rapport à cette instruction de saut) dans le segment de code (adresse de saut codée sur 8 bits).

Saut rapproché (*NEAR jump*): C'est un saut intra-segment: l'adresse de saut est située à l'intérieur du segment de code, de 32768 octets en arrière à 32767 octets en avant (adresse de saut codée sur 16 bits).

Saut éloigné (*FAR jump*): Un *FAR* jump est un saut inter-segment: l'adresse du saut se situe n'importe où dans la mémoire accessible et pas forcément dans le segment de code.

Saut direct: Dans le saut direct (ou absolu), l'adresse du saut est indiquée dans l'instruction elle-même.

Saut indirect: L'adresse de saut est dans un registre ou dans une case mémoire. Elle ne sera effectivement connue qu'au moment de l'exécution.

Saut inconditionnel: Une instruction de saut inconditionnel transfèrera automatiquement l'exécution à l'adresse de saut indiquée.

Saut conditionnel: Une instruction de saut conditionnel transfère l'exécution à l'adresse de saut selon la valeur des indicateurs: *overflow*, *carry*, *sign*, *zero*, *parity*. Si l'indicateur concerné n'est pas positionné, l'exécution continue à l'instruction qui suit l'instruction de saut.

V.7.1 Les instructions de saut inconditionnel.

L'instruction de base est **JMP**.

Exemple: *JMP Cas1* ;saut direct

JMP [*BX*] ;saut indirect à l'adresse contenue dans BX

V.7.2 Les instructions de saut conditionnel.

Ces instructions sont réparties en 3 groupes: saut sur valeur de flag, saut sur test arithmétique signée, saut sur test arithmétique non signée.

V.7.2.1 Les instructions de saut testant un flag.

JC / JNC Jump if Carry / not Carry

JS / JNS Jump if Sign/ not Sign

JO / JNO Jump if Overflow / not Overflow

JP / JNP	Jump if Parity / not Parity
JZ / JNZ	Jump if Zero / not Zero

V.7.2.2 Les instructions de saut sur test arithmétique signé.

JE	Jump if Equal
JNE	Jump if Not Equal
JG	Jump if Greater
JGE	Jump if Greater or Equal
JL	Jump if Less
JLE	Jump if Less or Equal

V.7.2.3 Les instructions de saut sur test arithmétique non signé.

JA	Jump if Above
JAE	Jump if Above or Equal
JB	Jump if Below
JBE	Jump if Below or Equal

V.8 Les instructions de boucle: LOOP, LOOPE et LOOPNE.

Ces instructions permettent de programmer les structures itératives:

do { instructions } While (condition), While (condition) {instructions},...

Elles utilisent le registre *CX* comme décompteur. Aussi il est recommandé de ne pas modifier *CX* à l'intérieur de la boucle. Ce registre *CX* doit être initialisé à une valeur positive avant l'entrée dans la boucle.

Syntaxe:	*LOOP*	*etiquette*
	LOOPE	*etiquette*
	LOOPNE	*etiquette*

LOOP décrémente le contenu de *CX* et le compare à 0; si *CX* est encore positif, il y a branchement à *étiquette*. *LOOPE* décrémente le contenu de CX et le compare à

0; si *CX* est encore positif et le flag *Z* est à 1, il y a branchement à *étiquette*. *LOOPNE* décrémente le contenu de *CX* et le compare à 0; si *CX* est encore positif et le flag *Z* est à 0, il y a branchement à *étiquette*.

Exemples 1:

```
_sommation  proc near
        MOV CX,100
Boucle:         MOV    AX,0
        ADD     AX,CX
        LOOP Boucle
_sommation endp
```

Ce petit exemple permet de faire la sommation des éléments de 0 à 100 en utilisant l'instruction LOOP.

L'équivalent de cette procédure en langage 'C' serait :

```
int near sommation(void) {
int x=0;
_CX=100;
do {
x=x+_CX;
_CX=_CX-1;
} while (_CX!=0);
return(x);
}
```

V.9 Les instructions sur chaînes d'octets.

Ce sont les instructions:

MOVS* (ou *MOVSB*, *MOVSW*), *CMPS

SCAS*, *LODS* et *STOS

V.9.1 MOVS (Move String).

Syntaxe: *MOVS chaîne-destination,chaîne-source*

Fonction: transfère un octet ou un mot de la *chaîne-source* (adressée par *SI*) vers la *chaîne-destination* (adressée par *DI*) et met à jour *SI* et *DI* pour que ces registres pointent vers l'élément suivant de leurs chaînes respectives. Quand *MOVS* est utilisée avec *REP*, elle permet le transfert d'une zone mémoire vers une autre zone mémoire.

V.9.2 MOVSB et MOVSW.

Syntaxe: *MOVSB*

MOVSW

Fonction: Ces instructions ont la même fonction que *MOVS*.

Elles ont l'avantage de spécifier l'unité de transfert: octet ou mot (16 bits). Une autre différence par rapport à *MOVS* est que *MOVSB* et *MOVSW* n'ont pas d'arguments: la chaîne source est adressée par *SI* et la chaîne destination par *DI*. Ces instructions supposent que la chaîne source est adressée relativement à *DS* et celle de destination est adressée relativement à *ES*. Nous donnons ci-dessous un exemple qui permet d'expliquer un peu mieux le fonctionnement de ces instructions. Dans cet exemple le programme permet de faire le transfert des éléments du tableau d'octets initial_1 vers le tableau du même type table_1. Ce programme fait également le transfert des éléments du tableau de mots initial_2 vers le tableau du même type table_2.

```
.model small

.data

table_1    db  5 dup(?)

table_2    dw  4 dup(?)

initial_1  db   1,2,5,4,9
```

```
initial_2  dw   1000,1002,1005,1008
.code
programme proc near
MOV ax,@data
mov ds,ax
mov es,ax
mov cx,5                        ; 5 élément à transferer
cld                             ; transfert dans le sens croissant
mov si,offset initial_1         ; adresse du tableau source dans SI
mov di,offset table_1           ; adresse du tableau destination dans DI
rep  movsb                      ; transfert des éléments de initial_1 vers table_1
mov     cx,4                    ; nombre d'éléments à transferer
mov si,offset initial_2         ; adresse du tableau source dans SI
mov di,offset table_2           ; adresse du tableau destination dans DI
rep movsw                       ; transfert des éléments de initial_2 vers table_2
mov     ax,4c00h
int     21h                     ; Fin du programme et retour au système d'exploitation
programme endp
end     programme
```

V.9.3 L'instruction CMPS.

Syntaxe: *CMPS chaîne-destination,chaîne-source*

Fonction: *CMPS* (Compare String) soustrait l'octet ou le mot de la *chaîne-destination* (adressée par *ES:DI*) de l'octet ou du mot de la *chaîne-source* (adressée par *DS:SI*). Les indicateurs sont positionnés en fonction du résultat de la soustraction. Les opérandes restent inchangées. *SI* et *DI* sont mis à jour pour pointer vers l'élément

suivant de leur chaîne respective. Par exemple, si un *JG* (branchement si supérieur) est utilisé après *CMPS*, le branchement a lieu si l'élément de destination est plus grand que l'élément source. Si *CMPS* est préfixée par *REPE* ou *REPZ*, l'instruction est interprétée comme : "Tantque non *fin_de_chaîne* ET *éléments de chaîne égaux* Faire comparaison". La fin de chaîne est détectée quand *CX* = 0. Si *CMPS* est préfixée par *REPNE* ou *REPNZ*, l'instruction est interprétée comme : " Tantque non *fin_de_chaîne* ET *éléments de chaîne différents* Faire comparaison " .

V.9.4 L'instruction SCAS.

Syntaxe: *SCAS chaine-destination*

Fonction: *SCAS* (Scan String) soustrait l'élément de la *chaine-destination* (octet ou mot) adressé par *DI* du contenu de *AL* (s'il s'agit d'une chaîne d'octets) ou du contenu de *AX* (s'il s'agit d'une chaîne de mots) puis met à jour les indicateurs en fonction du résultat de cette soustraction. La *chaîne-destination* n'est pas modifiée de même que le registre *AX*. De plus *SCAS* met à jour *DI* pour qu'il pointe vers l'élément suivant de *chaîne-destination*. Si *SCAS* est préfixée par *REPE* ou *REPZ*, l'instruction est interprétée comme: "Tantque non *fin_de_chaîne* ET *élément de chaîne=valeur du registre* (*ZF*=1) Faire *SCAS*". Si *SCAS* est préfixée par *REPNE* ou *REPNZ*, l'instruction est interprétée comme: "Tant que non *fin_de_chaîne* ET *élément de chaîne=valeur du registre* (*ZF*=0) Faire *SCAS* ".

V.9.5 L'instruction LODSB ou LODSW.

Syntaxe: *LODS* **chaîne-source**

Fonction: *LODS* (Load String) transfère l'élément de chaîne (octet ou mot) adressé par *SI* vers *AL* (*LODSB*) ou vers *AX* (*LODSW*); puis met à jour *SI* pour qu'il pointe vers l'élément prochain de la chaîne.

V.9.6 L'instruction STOSB ou STOSW.

Syntaxe: *STOS chaîne-destination*

Fonction: *STOS* (Store String) transfère un octet ou un mot contenu dans *AL* (*STOSB*) ou dans *AX* (*STOSW*) vers la chaîne-destination puis met à jour *DI* pour qu'il pointe vers l'élément suivant de la *chaîne-destination*.

VI. Notion de procédure.

La notion de procédure en assembleur correspond à celle de fonction en langage

C, ou de sous-programme dans d'autres langages. Dans l'exemple ci dessous, la procédure est nommée **calcul**. Après l'instruction B, le processeur passe à l'instruction C de la procédure, puis continue jusqu'à rencontrer RET et revient à l'instruction D. Une procédure est une suite d'instructions effectuant une action précise, qui sont regroupées par commodité et pour éviter d'avoir à les écrire à plusieurs reprises dans le programme. Les procédures sont repérées par l'adresse de leur première instruction, à laquelle on associe une étiquette en assembleur.

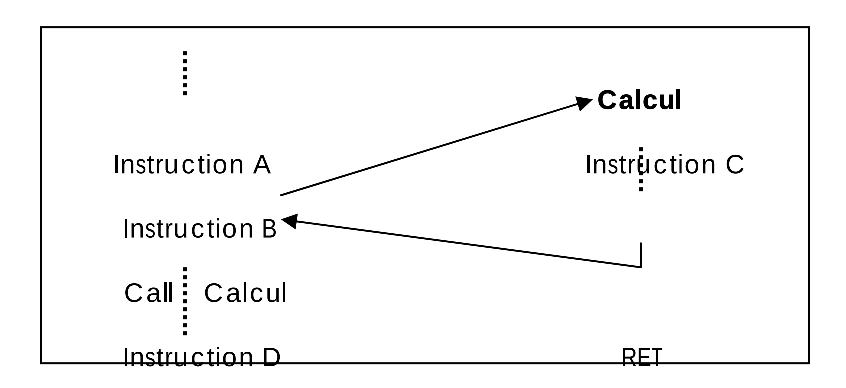


Figure 3.3 Appel d'une procédure.

L'exécution d'une procédure est déclenchée par un programme *appelant*. Une procédure peut elle-même appeler une autre procédure, et ainsi de suite.

VI.1 Instructions CALL et RET.

L'appel d'une procédure est effectué par l'instruction CALL. CALL *adresse_debut_procedure*

L'adresse peut être sur 16 bits, la procédure est donc dans le même segment d'instructions ou sur 32 bits lors d'un appel inter-segment (Far Call). La fin d'une procédure est marquée par l'instruction RET. Le processeur passe alors à l'instruction placée immédiatement après le CALL. RET est aussi une instruction de branchement : le registre IP (pointeur d'instructions) est modifié pour revenir à la valeur qu'il avait avant l'appel par CALL. On peut avoir un nombre quelconque d'appels imbriqués, comme sur la figure ci dessous. L'*adresse de retour*, utilisée par RET, est en fait sauvegardée sur la pile par l'instruction CALL. Lorsque le

processeur exécute l'instruction RET, il dépile l'adresse sur la pile (comme POP), et la range dans IP.

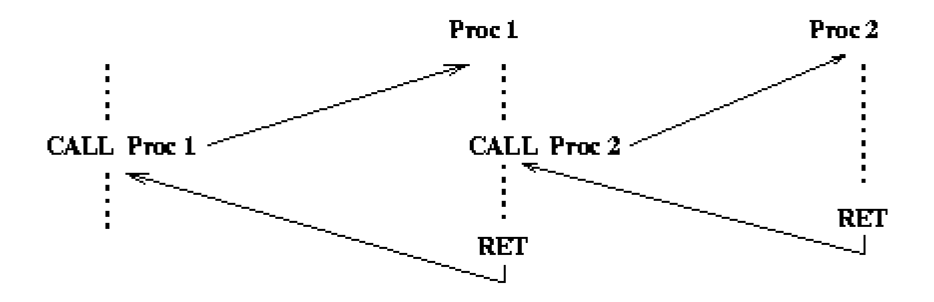


Figure 3.4 Plusieurs appels de procédures imbriqués.

L'instruction CALL effectue donc les opérations :

- Empiler la valeur de IP. A ce moment, IP pointe sur l'instruction qui suit le CALL.
- Placer dans IP l'adresse de la première instruction de la procédure (donnée en argument).

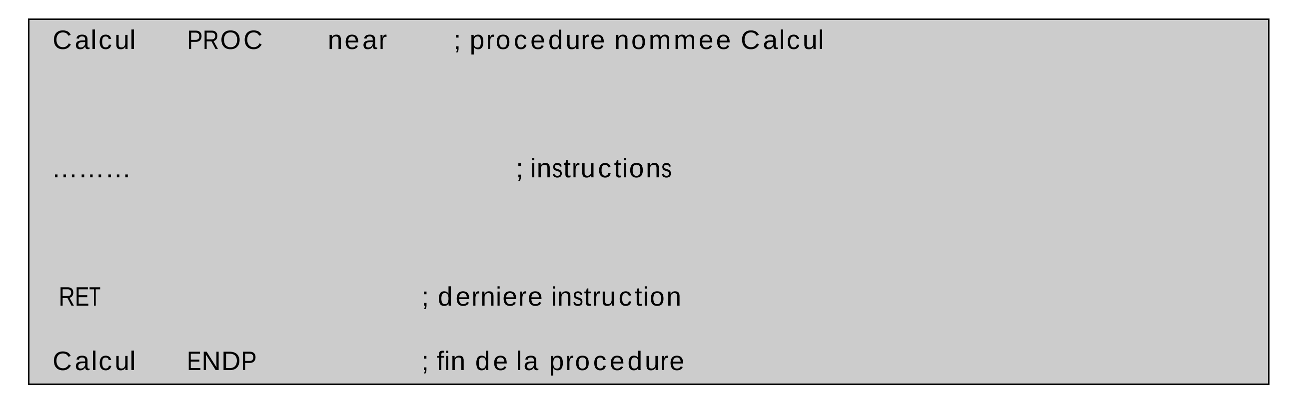
Et l'instruction RET :

- Dépiler une valeur et la ranger dans IP.

VI.2 Déclaration d'une procédure.

L'assembleur possède quelques directives facilitant la déclaration de procédures.

On déclare une procédure dans le segment d'instruction comme suit :



```
Calcul    PROC    near     ; procedure nommee Calcul

.........                       ; instructions

 RET                        ; derniere instruction
Calcul    ENDP              ; fin de la procedure
```

Le mot clef PROC commence la définition d'une procédure, near indiquant qu'il s'agit d'une procédure située dans le même segment d'instructions que le programme appelant. L'appel s'écrit simplement :

CALL Calcul

En général, une procédure effectue un traitement sur des données (*paramètres)* qui sont fournies par le programme appelant, et produit un résultat qui est transmis à ce programme. Plusieurs stratégies peuvent être employées :

1. Passage par registre *:* les valeurs des paramètres sont contenues dans des registres du processeur. C'est une méthode simple, mais qui ne convient que si le nombre de paramètres est petit (il y a peu de registres).

2. Passage par la pile *:* les valeurs des paramètres sont empilées. La procédure lit la pile.

VI.2.1 Passage de paramètres par registres.

Exemple avec passage par registre : On va écrire une procédure "SOMME" qui calcule la somme de 2 nombres naturels de 16 bits. Convenons que les entiers sont passés par les registres AX et BX, et que le résultat sera placé dans le registre AX. La procédure s'écrit alors très simplement :

```
SOMME PROC near  ; AX <- AX + BX

ADD AX, BX

RET

SOMME ENDP
```

et son appel, par exemple pour ajouter 6 à la variable var :

```
MOV  AX, 6

MOV  BX, var

CALL SOMME

MOV  var, AX
```

VI.2.2 Passage de paramètres par piles.

Exemple avec passage par la pile : Cette technique met oeuvre un nouveau registre, BP (*Base Pointer*), qui permet de lire des valeurs sur la pile sans les dépiler ni modifier SP. Le registre BP permet un mode d'adressage indirect spécial, de la forme :

MOV AX, [BP+6]

Cette instruction charge le contenu du mot mémoire d'adresse **BP+6** dans **AX**. Ainsi, on lira le sommet de la pile avec :

```
MOV  BP, SP      ; BP pointe sur le sommet
MOV  AX, [BP]    ; lit sans depiler
```

et le mot suivant avec :

MOV AX, [BP+2] ; 2 car 2 octets par mot de pile.

L'appel de la procédure "SOMME2" avec passage par la pile est :

PUSH 6

PUSH var

CALL SOMME2

La procédure SOMME2 va lire la pile pour obtenir la valeur des paramètres. Pour cela, il faut bien comprendre quel est le contenu de la pile après le CALL :

SP → IP (adresse de retour)

SP+2 → var (premier paramètre)

SP+4 → 6 (deuxième paramètre)

Le sommet de la pile contient l'adresse de retour (ancienne valeur de IP empilée par CALL). Chaque élément de la pile occupe deux octets.

La procédure SOMME2 s'écrit donc :

```
SOMME2   PROC near        ; AX <- arg1 + arg2

MOV  BP, SP               ; adresse sommet pile
MOV  AX, [BP+2]           ; charge argument 1

ADD  AX, [BP+4]           ; ajoute argument 2

RET

SOMME2  ENDP
```

La valeur de retour est laissée dans AX. La solution avec passage par la pile parait plus lourde sur cet exemple simple. Cependant, elle est beaucoup plus souple dans le cas général que le passage par registre. Il est très facile par exemple d'ajouter deux paramètres supplémentaires sur la pile. Une procédure bien écrite modifie le moins de registres possible. En général, l'accumulateur est utilisé pour transmettre le résultat et est donc modifié. Les autres registres utilisés par la procédure seront normalement sauvegardés sur la pile. Voici une autre version de SOMME2 qui ne modifie pas la valeur contenue par BP avant l'appel :

```
SOMME2   PROC   NEAR      ; AX <- arg1 + arg2
```

```
PUSH BP                  ; sauvegarde BP
MOV  BP, SP              ; adresse sommet pile

MOV  AX, [BP+4]          ; charge argument 1

ADD  AX, [BP+6]          ; ajoute argument 2

POP  BP                  ; restaure ancien BP

RET

SOMME2  ENDP
```

Noter que les index des arguments (BP+4 et BP+6) sont modifiés car on a ajouté une valeur au sommet de la pile.

VI.3 Traduction en assembleur du langage C sur PC.

Nous nous intéressons dans cette section à la traduction en assembleur des programmes en langage C sur PC (processeurs de la famille 80x86 que nous avons un étudié auparavant). Le détail de cette traduction (ou compilation) dépend bien entendu du compilateur utilisé et du système d'exploitation (DOS, Windows, UNIX,...). Il dépend aussi de divers réglages modifiables par le programmeur : taille du type **int** (16 ou 32 bits), modèle de mémoire utilisé (pointeurs sur 16 ou 32 bits, données et code dans des segments différents ou non, etc.).

Nous n'aborderons pas ces problèmes dans ce cours (voir la documentation détaillée du compilateur utilisé si besoin), mais nous étudierons quelques exemples de programmes C et leurs traductions. Le compilateur utilisé est Turbo C++ version 3 (en mode ANSI C) sous DOS, avec des entiers de 16 bits et le modèle de mémoire "small". Normalement, ce compilateur génère directement du code objet (fichier .OBJ) à partir d'un fichier source en langage C (fichier .C ou .CPP). Il est cependant possible de demander l'arrêt de la compilation pour obtenir du langage assembleur (fichier .ASM). Pour cela, utiliser sous DOS la commande :

C:\tc\bin\> tcc –S exemple.c

Un fichier **exemple.c** est alors créé. Considérons le programme en langage C suivant :

```
/* Programme EXEMPLE_1.c en langage C */


void main(void) {
```

```
char X = 11;

char C = 'A';
int Res;

if (X < 0)

  Res = -1;

else

  Res = 1;

}
```

Trois variables, **X, C** et **Res** sont définies avec ou sans valeur initiale. Ensuite, on teste le signe de X et range 1 ou -1 dans la variable **Res**.

Le programme suivant montre la traduction en assembleur effectuée par Turbo C.

```
_TEXT  SEGMENT byte public 'CODE'

;

;   void main(void) {

;

ASSUME    cs:_TEXT

_main  PROC      near

PUSH      bp

MOV       bp,sp

SUB       sp, 4

;

;           char X = 11;

;

MOV       byte ptr [bp-1], 11

;

;           char C = 'A';

;

MOV       byte ptr [bp-2], 65

;

;           int Res;
```

```
;           if (X < 0)

;

CMP      byte ptr [bp-1], 0

JGE      @1@86

;

;                 Res = -1;

;

MOV      word ptr [bp-4], 65535

JMP      @1@114

@1@86:

;

;           else

;                 Res = 1;

;

MOV      word ptr [bp-4], 1

@1@114:

;

;   }
;

MOV      sp,bp

POP      bp

RET

_main  ENDP

_TEXT  ENDS

END
```

Remarquons les points suivants :

1. La fonction `main()` est considérée à ce stade comme une procédure ordinaire (**`PROC near`**). C'est plus tard, lors de la phase d'édition de lien, qu'il sera indiqué que la fonction main()

correspond au point d'entrée du programme (première instruction à exécuter). La fonction est terminée par l'instruction **RET**.

2. On n'utilise pas ici de segment de données : toutes les variables sont allouées sur la pile.

3. L'allocation des variables sur la pile s'effectue simplement en soustrayant au pointeur **SP** le nombre d'octets que l'on va utiliser (ici **4**, car **2 variables X et C** d'un octet, plus une variable **Res** de **2 octets**).

4. La ligne X = 11 est traduite par :

```
MOV byte ptr [bp-1], 11
```

Noter l'utilisation de byte ptr pour indiquer que BP contient ici l'adresse d'une donnée de taille octet. Le test if (X <0) est traduit par une instruction CMP suivie d'un branchement conditionnel, utilisant une étiquette placée par le compilateur (d'où son nom étrange : @1@114). Chaque langage de programmation doit définir une convention de passage des paramètres lors des appels de procédures ou de fonctions. Cette convention permet de prévoir l'état de la pile avant, pendant et après un appel de fonction (dans quel ordre sont empilés les paramètres ? Qui est responsable de leur dépilement ? Comment est passée la valeur de retour ?). Etudions à partir d'un exemple simple comment sont passés les paramètres lors des appels de fonctions en langage C.

```
/* Programme EXEMPLE_2.C */
int ma_fonction( int x, int y ) {
  return x + y;
}

void main(void) {
  int X = 11;
  int Y = 22;
  int Res;
  Res = ma_fonction(X, Y);
}
```

La traduction en assembleur de ce programme (effectuée par Turbo C) donne le programme suivant :

```
_TEXT   SEGMENT byte public 'CODE'
;
; int ma_fonction( int x, int y ) {

ASSUME  cs:_TEXT
_ma_fonction    PROC    near
PUSH    bp
MOV     bp,sp
```

```
;
; return x + y;
;
MOV     ax, [bp+4]
ADD     ax, [bp+6]
;  }
POP     bp
RET
_ma_fonction    ENDP
;
; void main(void) {
;
ASSUME  cs:_TEXT
_main     PROC    near
PUSH    bp
MOV     bp,sp
SUB     sp,6
; int X = 11;
MOV     [bp-2], 11
; int Y = 22;
MOV     [bp-4], 22
;
; int Res;
; Res = ma_fonction(X, Y);
PUSH    word ptr [bp-4]
PUSH    word ptr [bp-2]
CALL    _ma_fonction
ADD     sp, 4
MOV     [bp-6],ax
; }
MOV     sp,bp
POP     bp
RET
_main     ENDP
_TEXT     ENDS
```

En étudiant cet exemple, on constate que :

1. la fonction **C ma_fonction()** a été traduite par une procédure assembleur nommée **_ma_fonction**, qui lit ses arguments sur la pile à l'aide de la technique que nous avons vue plus haut.

2. la fonction ne modifie pas l'état de la pile;

3. Avant l'appel de la fonction (**CALL**), les arguments sont empilés (**PUSH**). Après le retour de la fonction, le pointeur **SP** est incrémenté pour remettre la pile dans son état précédent (**ADD sp, 4** est équivalent à deux instructions **POP 2 octets**).

4. La valeur retournée par la fonction est passée dans **AX** (d'où l'instruction **MOV [bp-6], ax**).

Le respect des conventions d'appel de procédures est bien entendu très important si l'on désire mélanger des fonctions C et des procédures en assembleur.

VII. Les exceptions & Interruptions.

VII.1 Introduction.

Nous étudions dans cette partie les interruptions *matérielles* (ou externes), c'est à dire déclenchées par le matériel (hardware) extérieur au processeur. Les interruptions permettent au matériel de communiquer avec le processeur. On peut distinguer deux sortes d'interruptions. Les interruptions matérielles et les interruptions logicielles. Les interruptions internes, dues à l'exécution du programme (division par zéro, dépassement de capacité d'un registre, tentative d'accès à une zone mémoire protégée, ..) sont considérées comme interruptions logicielles. Ces interruptions sont souvent appelées exceptions. Les interruptions ainsi que les exceptions sont toutes traitées de la même façon. La différence vient surtout de la source d'enclenchement de l'interruption. En effet les interruptions matérielles et les exceptions peuvent se produire n'importe quand, tandis que les interruptions logicielles se produisent à un endroit précis du code où se trouve une des instructions **int** ou **into** ou **trap** (ça dépend du microprocesseur). De plus, les interruptions matérielles peuvent être masquées (interdites ou autorisées) à l'aide de l'indicateur **IF** (Registre d'état). Une interruption externe est signalée au processeur par un signal électrique sur une des ses pattes prévues pour ceci. Lors de la réception de ce signal, le processeur "traite" l'interruption dès la fin de l'instruction en cour d'exécution. Ce traitement consiste à exécuter un programme qui est appelé automatiquement lorsque l'interruption survient. L'adresse de début du programme doit être préalablement stockée dans une table dite des ***vecteurs d'interruptions***. Lorsque la routine d'interruption est exécutée une instruction ***IRET*** permet au processeur de reprendre l'exécution à l'endroit où il avait été interrompu. Il est aussi possible d'ignorer l'événement et passer normalement à l'instruction suivante dans le cas des ***interruptions masquables***. Il est en effet parfois nécessaire de pouvoir ignorer les interruptions pendant un certain temps, pour effectuer des traitements plus urgents. Lorsque le traitement est terminé, le processeur *démasque* les interruptions et les prend alors en compte. La procédure d'interruption est donc une tache spéciale associée à un événement externe ou interne.

VII.2 Les interruptions matérielles.

VII.2.1 Cas des processeurs de la famille INTEL.

Les interruptions matérielles sont générées par les périphériques : souris, clavier, disque, horloge temps réel, etc. À la différence des interruptions logicielles, elles peuvent être autorisées ou interdites au moyen de l'indicateur **IF** du registre d'état «**EFLAGS** ». Comme le Pentium n'a que deux entrées d'interruption matérielle, **NMI** et **INTR**, on doit lui adjoindre un contrôleur d'interruption programmable afin de pouvoir disposer de plusieurs sources d'interruption avec une gestion de priorité. C'est le PIC (Programmable Interrupt Controler 8259A).

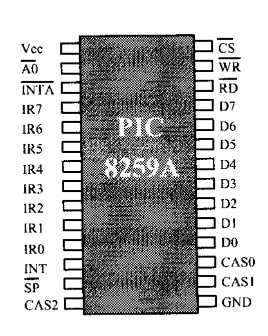


Figure 3.5 Le contrôleur d'interruption PIC 8259A.

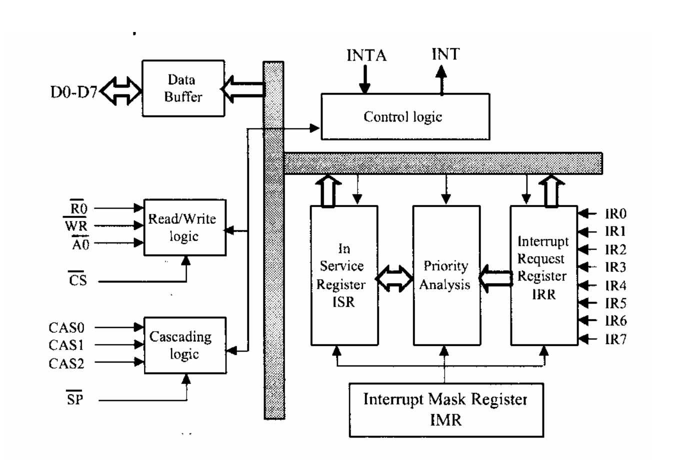


Figure 3.6 Schéma interne du PIC 8259.

Le 8259A peut accepter les interruptions de 8 sources externes, et on peut gérer jusqu'à 64 sources différentes en cascadant plusieurs 8259A. Il gère la priorité entre les interruptions simultanées, interrompt le processeur et lui passe un code pour identifier la source d'interruption. Une source d'interruption est connectée à chacune des 8 entrées IR0 à IR7. Selon sa priorité, et s'il n'y a pas d'autre interruption en cour, le PIC décide s'il peut transmettre l'interruption au CPU. Si oui, il affirme la ligne INT, qui est connectée à l'entrée INTR (Interrupt Request) du CPU. Si le CPU est prêt à

accepter l'interruption, il répond au PIC via la ligne INTA (Interrupt Acknowledge). Le PIC répond à son tour en envoyant le numéro d'interruption sur les lignes D0 à D7. Ce numéro est un index dans la table des vecteurs d'interruption. Le CPU est maintenant prêt à appeler le sous-programme de traitement d'interruption approprié. Quand le sous-programme de traitement d'interruption a terminé son exécution, il en avertit le PIC pour qu'il puisse permettre à d'autres interruptions d'atteindre le CPU. Les interruptions matérielles servent à une gestion efficace des périphériques d'entrée/sortie. Dans un ordinateur moderne, il y a continuellement des interruptions matérielles. Le temporisateur, l'horloge temps réel, les touches du clavier, les mouvements et les clics de la souris, le modem, l'imprimante, les disques durs et souples, le CDRom, sont tous des sources d'interruptions. Les circuits contrôleurs de périphériques contiennent plusieurs registres d'interface avec le CPU. Il y'a habituellement un registre de contrôle, un registre d'état, et un ou plusieurs registres de données. Pour connaître l'état d'un périphérique, le CPU peut interroger le registre d'état. L'approche des drapeaux (flags) consiste à interroger de façon répétitive le registre d'état, pour savoir où le périphérique est rendu dans le transfert des données. A-t-il reçu une nouvelle donnée ? A-t-il terminé la transmission de la dernière donnée envoyée ? etc. Cette approche consomme trop de temps de la part du processeur. L'approche interruption est beaucoup plus performante. Le périphérique envoie une interruption matérielle au processeur quand il a quelque chose à signaler. Le processeur interrompt alors la tâche en cour, enregistre en mémoire l'état de la machine, et vient interroger le registre d'état du périphérique, pour connaître la cause de l'interruption. Il effectue ensuite le traitement approprié et élimine la source de l'interruption Ce traitement consiste, par exemple, à lire la donnée reçue dans le registre de réception et à l'inscrire en mémoire, ou à lire en mémoire la prochaine donnée à transmettre et à l'inscrire dans le registre de transmission du périphérique. Le processeur retourne ensuite à la tâche interrompue après avoir restauré l'état de la machine qu'il avait enregistré au moment de l'interruption.

Le sous-programme de traitement a donc 4 tâches à exécuter :

-Sauvegarder l'état de la machine en empilant les registres susceptibles d'être modifiés dans le sous-programme de traitement d'interruption (ISR). Ceci inclut EFLAGS (registre d'état);

-Interroger le registre d'état d u périphérique pour savoir quelle opération effectuer ;

-Éliminer la source de l'interruption en effectuant l'opération d'entrée-sortie ;

-Restaurer l'état de la machine et retourner à la tâche interrompue en dépilant les registres empilés.

VII.2.2 PIC dans le cas du PC.

L'architecture est partiellement figée dans le cas des PC (historique par rapport au ×286). Le PIC maître va vers le processeur et le PIC esclave sur la broche 2 du maître.

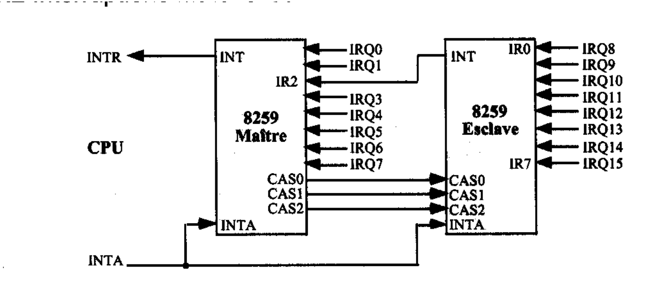


Figure 3.7 Cascade de deux contrôleurs d'interruptions PIC 8259.

Depuis le premier 8088 qui a équipé les PC, les **interruptions** sont normalisées pour les périphériques standard. La dernière composante d'un système à microprocesseur est l'accès direct à la mémoire, le **DMA** (Direct Memory Access). Cette fonction permet de demander au processeur de se déconnecter des bus de données, adresse et contrôle pendant qu'un périphérique prend le contrôle de la mémoire. L'arrivée du pseudo Plug & Play de Windows 95/98 et Win2000 couplé avec le plug & Play du Bios facilite dans certains cas le paramétrage des interruptions, mais il est parfois nécessaire de reprendre les configurations manuellement lorsque le nombre de périphériques augmente. Un périphérique est donc déterminé de manière hardware par une plage d'**adresses** (parfois une seule adresse), **une interruption** et éventuellement un canal **DMA**. Le 8088-8086 utilisait un 8259 d'INTEL, avec 8 niveaux d'interruptions notées de IRQ0 à IRQ7. Le 80286 jusqu'aux 486 utilisaient 2 contrôleurs 8259 chaînés ce qui permet d'avoir 16 sources d'interruptions. Ils sont notés de IRQ0 à IRQ15, mais avec l'IRQ9 réservée pour le chaînage. Depuis les Pentium (et un nouveau circuit gestionnaire d'interruptions), cette IRQ9 est accessible aux périphériques, mais ceci explique que peu de périphériques ne se branchent par défaut sur ce niveau d'interruption. L'interruption non masquable (**NMI**) est également utilisée pour des erreurs de parité ou erreurs de connecteurs entrées / sorties. Le tableau ci-dessous reprend l'ensemble des adresses et interruptions normalisées du PC, mélangées avec les interruptions utilisées par défaut

actuellement. Souvent les interruptions 14 et 15 sont utilisées par le contrôleur de disque dur (une par port IDE). L'interruption 12 est utilisée par la souris PS2. Le port USB utilise également par défaut cette interruption 12. Néanmoins, le port USB passe en 11 si une souris PS2 est connectée. Ces interruptions peuvent être vérifiées par les outils fournis par les systèmes d'exploitation et pour certaines, elles sont affichées lors du démarrage de la majorité des PC. La vérification dans les PC sous DOS passe par le programme **MSD.exe** situé en c:\DOS. Dans le cas de Windows cet utilitaire est situé dans le panneau de configuration. Il permet les mêmes vérifications, mais de manière plus étendues. Depuis Win98, un nouvel utilitaire, Outils systèmes Microsoft**,** situé dans le dossier Outils Système du menu Démarrer permet une meilleure connaissance sur le système en cour. En regardant ce tableau, si un PC manque d'interruptions libre, il est possible d'en récupérer en supprimant des ports de communication. Si vous n'utilisez pas de ports série (uniquement USB), vous pouvez rendre inactif COM 1 et Com 2 (disabled) et récupérer les interruptions 3 et 4. Idem pour le port parallèle. Si vous utilisez 2 ports parallèles, vous pouvez mettre l'interruption de LPT2 en 5 (selon la norme) ou en 7. Si vous utilisez des disques SCSI, pourquoi pas rendre inactif les ports IDE 1 et IDE2 dans le BIOS (récupération des INT 14 et 15), ... Comme les systèmes

Fonction	IRQ	Adresse	Commentaires
Horloge système	0		Ne peut être modifié
Clavier	1		Ne peut-être modifié
Contrôleur d'IRQ programma	2		Ne peut être modifié
Com 2 Com 4	3	2F8-2FF 2E8 - 2EF	Modifiable avec prudence
Com 1 Com 3	4	3F8-3FF 3E8-3EF	Modifiable avec prudence
Libre	5		Généralement utilisé par la carte son, également LPT2 en 278
Contrôleur lecteur de disque	6		Modifiable avec prudence
LPT 1 LPT 2	7	378 –37A (pour les produits de	Modifiable avec prudence. On peut mettre LPT1 et LPT2 sur la même interruption, mais pas d'impressions simultanées

		marque) 278 – 27A	
Horloge: date et heure	8		Pas modifiable
	9		Libre en Pentium et suivant, utilisé en parallèle avec IRQ2 dans les 486. Souvent utilisé par les cartes sons
	10 (A)		Libre
	11 (B)		Libre
	12 (C)		Libre
Coprocesseur mathématique	13 (D)		Ne peut être modifié
	14 (E)		Libre, généralement IDE 1
	15 (F)		Libre, généralement IDE 2

Tableau 2. Ensemble des interruptions normalisées du PC

d'exploitation Windows NT, 2000 et XP sont des systèmes d'exploitation pleinement protégés, les application: en mode utilisateur ne peuvent accéder au matériel directement et doivent passer par un pilote de périphérique fonctionnant en mode noyau. C'est la couche **HAL** (HardwareAbstraction Layer) de NT qui gère les interruptions. La réalisation de tels pilotes de périphériques dépasse toutefois le cadre de ce cours. Cependant, les **VDD** (Virtual Device Drivers) émulent les applications 16 bits de MS-DOS. Ils piègent ce que l'application MS-DOS croit être des références à des ports d'entrée-sortie et les traduisent en fonctions natives d'entrée-sortie Win32. L'application intitulée «Invite de commande» ou, en anglais, «DOS prompt», est un tel **VDD**. On peut y exécuter la plupart des programmes écrits pour MS-DOS, y compris certains qui utilisent des interruptions matérielles. C'est le cas des exemples qui suivent que vous pouvez compiler avec un assembleur tel que A86, MASM ou TASM. En mode protégé, le processeur va plutôt lire la table des descripteurs d'interruption IDT (Interrupt Descriptor Table). L'adresse de cette table est contenue dans le registre **IDTR**. Elle contient des descripteurs de 64 bits pour les sous-programmes de traitement d'interruption. Ces descripteurs sont appelés Trap Gates dans le cas d'interruptions logicielles. La procédure qui veut utiliser ces instructions doit avoir un niveau de privilège lui permettant d'accéder au descripteur d'interruption de la table **IDT** pour le numéro de trappe en question.

Ainsi, si une application de niveau 3 veut exécuter **int 47**, il faut que le descripteur à **IDT(47)** ait **DPL=3**. Dans un système d'exploitation tel que Windows NT, la **IDT** se trouve en mémoire protégée, de sorte qu'il n'est pas possible d'aller écrire un descripteur directement dans la **IDT**. Windows NT ne supporte pas non plus les **INT 0x21** de MS-DOS. Toutefois, Il est possible de les essayer en écrivant un programme assembleur et en l'exécutant dans la fenêtre «Invite de commande» (DOS prompt). En particulier, si nous désirons implanter notre propre sous-programme de traitement d'interruption logicielle, l'appel système **int 21h (33)** fonction **25h** permet de le faire facilement.

VII.3 Les interruptions logicielles.

Comme nous l'avions écrit plus haut, les interruptions logicielles, sont aussi appelées trappes ou déroutements. Elles incluent aussi les fautes et les arrêts. Une faute se produit quand le processeur détecte une erreur durant le traitement d'une instruction. Par exemple, **division par 0**, **opcode invalide**, etc. En mode réel (Mode 8086), quand le processeur rencontre une instruction telle que int immed8 (immed8 : constante sur un octet), il va lire la table des vecteurs d'interruption **IVT** (Interrupt Vector Table). Cette table de 1 Ko est située à l'adresse **0000:0000.** Chaque entrée de la table contient le numéro de segment de 16 bits et l'offset de 16 bits pour l'adresse d'un sous- programme de traitement d'interruption (Interrupt Service Routine ou **ISR**). Immed8 est utilisé comme indice dans cette table (on doit le multiplier par 4 pour avoir l'adresse physique correspondante). De plus, le registre d'état «FLAGS » et les registres de segment CS et d'instructions IP sont empilés dans cet ordre exact, puis les indicateurs **TF** et **IF** sont mis à 0. Le sous-programme d'interruption devra se terminer par l'instruction **IRET**, pour dépiler correctement ces paramètres. L'exécution se poursuit ensuite à l'adresse contenue dans la table. Par exemple, **int 8** ira lire le vecteur situé à l'adresse 32 (**0x20**) et branchera à l'adresse qui y est contenue. A cette adresse doit débuter un sous-programme de traitement d'interruption. Certaines ISR font partie du système d'exploitation et sont déjà définies, comme celles qui correspondent aux **INT 0x21** de MS-DOS. Cette trappe y sert de mécanisme d'appel au système d'exploitation. Si vous voulez définir votre propre sous-programme de traitement d'interruption, il suffit d'aller écrire son adresse à l'endroit approprié dans la table des vecteurs d'interruption avant de l'utiliser.

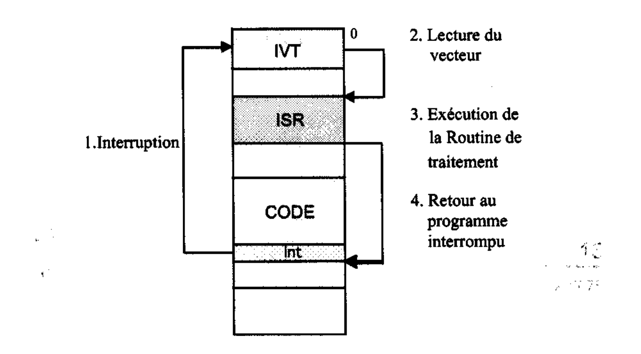


Figure 3.8 Principe d'exécution des interruptions

Exemple 1 : Interruption logicielle pour écrire une chaîne à l'écran :

```
mov ah,09
mov dx, offset string
int 33
```

où on a défini la chaîne de caractères string comme suit :

```
string db 'Toto':$'
```

Exemple 2: Interruption logicielle pour la lecture de l'horloge système.

L'horloge système interrompt le processeur **18,2 fois par seconde** si l'indicateur d'interruption IF est 1 (interruption matérielle 08). Chaque fois, le compteur de 32 bits situé aux adresses **0040:006C** et **0040:006E** est incrémenté de 1. Pour lire ce compteur, on peut utiliser l'interruption logicielle **int 0x1A** (26).

```
mov ah, 0      ; mode lecture de l'horloge
int 26         ; appel à l'interruption d'horloge de MS-DOS
```

Après l'instruction int 26,

cx = TIMER-HIGH = mot de poids fort du compteur

dx = TIMER-LOW = mot de poids faible du compteur

Si **al = 0**, on n'a pas dépassé **24h** depuis la dernière lecture

Si **al > 0**, on a dépassé **24h** depuis la dernière lecture.

Exemple 3: Interruption logicielle pour l'écriture d'un programme résident et déviation de l'interruption d'horloge temps réel.

Voici un exemple d'un programme résident qui utilise l'horloge temps réel et émet un bip chaque 10 sec. La procédure de traitement de l'interruption 8 est déviée pour exécuter notre sous-programme, puis continue avec l'ancienne procédure. Cette technique s'appelle un «hook».

```
start segment
org 100h
main: jmp short install
oldint8 dd ?                ; espace pour ancien vecteur
counter dw 182              ; 182 * 54,94 ms = 10 sec.
newint8 proc                ; nouvelISR pour interruption 8
dec cs:counter                      ; décrémenter compteur
jnz done                    ; on ne fait rien tant que count > 0
mov cs:counter,182          ; réinitialiser compteur
mov ah, 0Eh                 ; émettre un bip
mov al, 7
int 10h
done: jmp cs:oldint8        ; continuer dans l'ancien vecteur
newint8 endp
; la partie suivante ne sera exécutée qu'une fois, lors de l'installation
install proc near
mov ah, 35h                 ; charger CS:IP pour interruption 08
mov al, 08h                 ; IP va dans BX et CS dans ES
int 21 h
mov word ptr oldint8, bx    ; enregistrer dans espace prévu
```

```
mov word ptr oldint8+2, es
mov dx, offset newint8          ; ou lea dx, newint8
mov ah, 25h                     ; définir CS:IP pour nouvelle
mov al, 08h                     ; interruption 8
int 21h                    ; DX = IP, DS = CS par défaut
mov dx, (offset install -offset start) ; taille du progr. résident
add dx, 15                      ; arrondir à multiple de 16
shr dx, 4                       ; division par 16
mov ah,31h                      ; rendre résident
int 21 h
install endp
start ends
end main
```

VIII. Le compilateur assembleur.

Le compilateur assembleur traduit les programmes sources écrits en assembleur en leur équivalent binaire (le code machine). Cette opération nécessite plusieurs étapes, et il faut utiliser les services d'un lieur (linker) ou éditeur de liens (LINK.EXE, TLINK.EXE). L'éditeur de liens transforme le code généré par « rassembleur en code exécutable » (fichier d'extension EXE). Il peut aussi réaliser d'autres tâches qui ne nous intéresseront pas ici. Mais il se trouve que DOS connaît une deuxième forme de programme exécutable. Quoiqu'un peu ancienne, elle convient à merveille au langage machine. Il s'agit des fichiers COM. Pour obtenir des programmes COM, il faut disposer d'un utilitaire appelé EXE2BIN.EXE (sauf pour TASM). Précisons bien qu'EXE2BIN ne constitue pas une partie de rassembleur : il s'agit d'un utilitaire appartenant au système d'exploitation. Le programme système DEBUG peut aussi traduire du code écrit en assembleur et il est même capable de l'exécuter. Au point de vue confort d'utilisation, DEBUG est tout-à-fait limite. Par ailleurs DEBUG ne respecte pas intégralement les conventions mnémoniques des processeurs d'Intel (les mnémoniques sont les instructions en assembleur}. Nous allons donner un petit exemple de programmation avec DEBUG.

Un petit programme réalisé avec DEBUG :

Déclenchez le programme DEBUG en tapant au niveau du système d'exploitation DEBUG.

Un tiret apparaît à l'écran pour vous inviter à saisir des instructions. Tapez alors les lignes qui figurent ci-dessous. Chaque ligne doit se terminer par 'Enter'. Observez bien le résultat à l'écran. Pour le moment, il n'est pas important de comprendre le sens des instructions. Notre but est simplement de montrer la différence entre DEBUG et un assembleur normal.

```
C:\DOS>debug
-a
24A2 : 0100 ORG 200
24A2 : 0200 DB "Bien le bonjour de la part de DEBUG!$"
24A2 : 0225 ORG 100
24A2 : 0100 MOV AH,09
24A2 : 0102 MOV DX,200
24A2 : 0105 INT 21
24A2 : 0107 MOV AX,4C00
24A2 : 010A INT 21
24A2 : 010C  [Return]
-g
```

Voici en contrepartie le même programme écrit pour MASM/TASM :

```
DOSSEG
.MODEL SMALL
.STACK 50
.DATA
LIGNE DB "Bien le bonjour de la part de MASM/TASM!$"
.CODE
MOV AX,@DATA
```

```
MOV DS,AX
MOV AH,09H
MOV DX,OFFSET LIGNE
INT 21H
MOV AX,4C00H
INT 21H
END
```

MASM ou TASM

Ces deux compilateurs assembleurs sont compatibles entre eux. Chacun d'entre eux a ses avantages et ses inconvénients, ils ont le même prix et les mêmes performances. Le choix n'est donc pas évident. Toutefois si nous avons l'intention d'inclure des modules en assembleur dans des programmes écrits en langage évolué, il est préférable de prendre un assembleur de même origine que le langage évolué. Si vous travaillez déjà avec un langage de Borland (par ex: Turbo-Pascal ou Turbo C/C++), TASM sera le meilleur choix. Si par contre, vous programmez en Fortran ou en VB ou VC, il vaut mieux acheter MASM de Microsoft. MASM possède une interface très perfectionnée avec les langages évolués, ce qui facilite l'inclusion de modules en assembleur. Par ailleurs, MASM 6.0 est compatible avec OS/2. Si vous optez un jour pour ce système, vous n'avez pas besoin de vous procurer une mise à jour de l'assembleur. TASM est réputé pour sa vitesse de traduction. Quiconque possède déjà un langage Borland se sent immédiatement à son aise. Vous pouvez maintenant faire votre choix librement en toute connaissance de cause. Dans nos exemples nous nous servirons essentiellement du macro assembleur MASM de Microsoft. Il s'agit en effet de l'assembleur le plus répandu. Tous les autres sont généralement compatibles avec lui.

VIII.1 Les Directives de compilation.

Initialement, les directives étaient prévues pour simplifier la vie du programmeur, au fur et à mesure des versions successives des assembleurs, leur nombre s'est tellement accru que l'effet inverse commence à être perceptible, même en admettant que chacune d'elles a son utilité, on peut légitimement craindre d'être quelque peu submergé par la variété des possibilités offertes: La simple énumération des directives

disponibles suffirait à remplir quelques dizaines de pages. Leur nombre étant maintenant de loin supérieur à celui des instructions machine, mais la pratique quotidienne ne nécessite qu'un petit noyau de directives. Dans les paragraphes suivants. Nous allons passer en revue les directives les plus 'couramment' utilisées.

VIII.1.1 Directives de sélection du processeur.

En fonctionnement de base, les assembleurs MASM et TASM sont prévus pour traduire du code destiné aux microprocesseurs 8088 et 8086 et au coprocesseur arithmétique associé. Lorsqu'un système abrite un processeur Intel plus évolué, de la classe x86, il peut être intéressant d'adapter le code pour en exploiter les perfectionnements. Il faut alors demander à l'assembleur de mettre en service les instructions spéciales étendues. En pratique on insère dans le programme source une directive qui se compose d'un point et de la désignation abrégée du microprocesseur. Plus précisément, les directives suivantes sont à la disposition du programmeur :

.8086 : Active le jeu d'instructions du 8086 et 8088 et du coprocesseur 8087. C'est le paramétrage par défaut en l'absence de toute directive explicite de sélection du processeur.

.186 : Active le jeu des instructions du 80188 et 80186, y compris celles du coprocesseur 8087.

.286 : Active le jeu d'instructions du 80286 en mode réel, y compris les instructions du coprocesseur 80287.

.386 : Active le jeu des instructions du 80386 en mode réel, y compris les instructions du coprocesseur 80387.

.8087, .287, .387 : Activent uniquement le jeu d'instructions du coprocesseur arithmétique désigné sans se préoccuper du processeur.

Les instructions en mode protégé prévues pour le 80286 et 80386 et 80486 ne deviennent accessibles que si on accole la lettre "P" à la directive de sélection du processeur concerné (.286P, .386P, .486P). Toutes ces directives peuvent être utilisées dans n'importe quel module source. Chacune d'elles annule la précédente en vigueur.

VIII.1.2 Directives de sélection du modèle mémoire.

.Model SMALL: Ensemble le segment de code et le segment de données ne doivent pas dépasser 64 Ko mais ils ne commencent pas forcément à la même adresse. Toutes les adresses du programme sont de type court (NEAR).

.Model MEDIUM: La zone réservée aux données peut aller jusqu'à 64 Ko. Le code, c'est-à-dire le programme proprement dit, peut occuper de 0 à 640 Ko. Ne convient qu'aux programmes .EXE. L'accès aux données est réalisé par des adresses courtes (NEAR) tandis que le code contient des appels longs (FAR).

.Model COMPACT : C'est un peu l'inverse du modèle MEDIUM. Le code doit tenir dans 64 Ko. Mais la zone de données peut s'étendre de 0 à 640 Ko. Ne convient qu'aux programmes .EXE. L'accès au code est effectué par des appels courts (NEAR) tandis que les adresses des données sont systématiquement de type long (FAR).

.Model LARGE : Le code et les données peuvent occuper de 0 à 640 Ko. Ne convient qu'aux programmes .EXE. L'accès aux données et au code est uniquement assuré par des appels ou des adresses de type long (FAR).

.Model HUGE : Interprété de la même façon que LARGE.

.Model FLAT : Ce modèle mémoire est utilisé dans le cas d'une programmation 32 bits. C'est le modèle mémoire utilisé par Windows.

Pour les programmes en assembleur à vocation autonome, il est plus simple de choisir le modèle **SMALL**. Le code généré est alors rapide et efficace. Il faut dire qu'un programme écrit en assembleur dépasse rarement 64 Ko. Mais il ne suffit pas de définir le modèle mémoire souhaité, les données et le code doivent être rangés sous des directives **.DATA** et **.CODE**. Les assembleurs connaissent à ce sujet quelques variantes que nous allons évoquer brièvement. Le segment **.DATA ?** est prévu pour héberger des données non initialisées du type court (adresses sous forme d'offsets). En temps normal, ce segment ne sert que si le module en assembleur est destiné à être inclus dans un programme de haut niveau. Dans ce cas il sera rempli par des déclarations du type **DB (?)** et **DUP (?)**. Le segment **.FARDATA** est prévu pour recevoir des données initialisées accessibles par des adresses de type long (segment:offset). Normalement ce segment n'est utile que si le module en assembleur doit être inclus par la suite dans un programme de haut niveau. Le segment **.FARDATA ?** est prévu

pour stocker des données non initialisées accessibles par des adresses de type long (segment:offset). Normalement ce segment ne se justifie que si le module en assembleur doit être inclus par la suite dans un programme de haut niveau. Dans ce cas, il sera rempli par des déclarations du genre **DB (?)** ou **DUP (?)**. Les segments que nous venons de passer en revue sont rarement mis en service. La plupart des programmes de haut niveau peuvent être complétés par des modules exploitant uniquement les ressources des segments **.CODE** et **.DATA.** Nous donnons par la suite quelques exemples de déclaration de variables utilisant les segment **.DATA et .DATA ?**

-Déclaration de variables initialisées

```
.DATA
octet db -2
octets db 8 dup(0)
chaine db "Bonjour",0
mot dw 7FFFh
doubleMots dd 2 dup(0FFFFFFFFh)
nombre_reel real4      1.0
```

-Déclaration de variables non initialisées

```
.DATA ?
Mot dw ?                ; mot de 16 bits
Octets db 8 dup(?)      ; tableau de 8 octets
LENGTH equ 20                  ; définition d’une constante
dix equu 10                    ; définition d’une constante
```

VIII.1.3 Directives de décision.

Les structures de décision des langages de haut niveau que nous pouvons utiliser sont les suivants :

.IF *condition1*
instructions

[[**.ELSEIF** *condition2*
instructions]]

[[**.ELSE**
instructions]]

.ENDIF
Exemple:

```
.IF cx == 20
mov dx, 20
.ELSE
mov dx, 30
.ENDIF
```

Le code généré est le suivant (en gras):

```
.IF cx == 20
0017 83 F9 14    cmp cx, 014h
001A 75 05                jne @C0001
001C BA 0014  mov dx, 20
.ELSE
001F EB 03                jmp @C0003
0021 @C0001:
0021 BA 001E mov dx, 30
.ENDIF
0024 @C0003:
```

VIII.1.4 Directives de programmation structurée.

Les directives suivantes permettent de structurer le programme et de le rendre lisible et simple à comprendre.

.WHILE ... **.ENDW**

.REPEAT ... **.UNTIL**

.REPEAT ... .UNTILCXZ

.BREAK

Exemple :

```
.DATA
buf1 BYTE "This is a string",'$'
```

```
buf2 BYTE 100 DUP (?)
.CODE
sub bx, bx                ; mise à zero de bx
.WHILE (buf1[bx] != '$')
mov al, buf1[bx]          ; Lire un caractère

mov buf2[bx], al          ; le placer dans buf2

inc bx                    ; incrementation de bx

.ENDW
```

VIII.1.5 Directives de déclaration de procédures.

Nous ne pouvons pas donner dans le cadre de ce cours toutes les directives de déclarations de procédures. Leur nombre est assez élevé et leurs explications nécessitent une dizaine de pages. Toutefois nous donnons quelques exemples utilisant certaines directives afin de montrer leur intérêt ainsi que les possibilités qu'ils offrent au programmeur pour mieux structurer son programme et le rendre simple à comprendre et lisible. Nous donnons dans un premier exemple une procédure d'addition de trois variables arg1, arg2 et arg3. Nous supposons que ces variables sont passées à cette procédure en tant que paramètres. Nous avons volontairement utilisé dans cette procédure la directive **USES** qui permet de sauvegarder les registres modifiés par la procédure (ici di et si). Le code de la procédure écrit en gras est automatiquement généré par le compilateur.

Exemple :

addition PROC NEAR C USES di si,

arg1:WORD, arg2:WORD, arg3:WORD

```
push bp          ;sauvegarde de BP
mov bp, sp       ;faire pointer BP sur SP
push di          ;sauvegarde de di
push si          ;sauvegarde de si
mov     si,arg1
mov     di,arg2
mov     ax,arg3
add     ax,di
add     ax,si
pop si           ;récupération de si
pop di           ;récupération de di
mov sp, bp       ;remettre sp à sa valeur initiale
pop bp           ;récupération de bp
ret 6            ;nettoyage de la pile (6 parce qu'il y'a trois paramètres passés
                  ;à la procédure
addition endp
```

L'appel de cette procédure par le programme principal peut se faire simplement de la manière suivante:

INVOKE addition, 1, 2, 3

Il s'agit bien évidemment de l'addition des 3 nombres 1, 2 et 3. Le résultat de l'addition est renvoyé dans le registre AX.

VIII.2 Interfaçage entre l'assembleur et le langage 'C'.

Dans cette partie nous allons surtout insister sur l'appel de procédures 'C' à partir de l'assembleur. Le cas inverse ne pose pas de problèmes particuliers et a été déjà abordé dans le paragraphe VI. Dans l'exemple suivant nous utilisons certaines directives qui permettent d'accéder à partir de l'assembleur aux procédures 'C'. Nous avons pris comme exemple un appel à partir de l'assembleur de la procédure **printf**. Rappelons à cet égard que le prototype de la procédure printf est le suivant : int   Cdecl **printf**(const char ***__**format, ...);

Exemple:

```
.MODEL small, c                    ; Model mémoire ainsi que conventions d'appel c
.
printf PROTO NEAR,                 ; prototype de la fonction printf (VARAG: arguments variables)
pstring:NEAR PTR BYTE, num1:WORD, num2:VARARG.DATA

format BYTE '%i %i', 13, 0         ; chaîne de caractères à afficher par printf
.CODE
_main PROC PUBLIC                  ; Procédure assembleur qui fait appel à printf
.
INVOKE printf, OFFSET format, ax, bx
.
.
.
_main endp
END _main                                    ; Fin de la procédure
```

Dans le dernier exemple nous donnons un programme complet, avec les explications nécessaires, qui illustre bien l'interfaçage entre le langage 'C' et l'assembleur. En effet, il est intéressant parfois d'inclure des procédures 'C' dans un programme assembleur. Ceci a pour but d'optimiser le temps de programmation. Nous pouvons alors concentrer notre effort de programmation en assembleur sur les procédures à fortes contraintes temporelles (temps d'exécution le plus rapide). Il est tout à fait possible d'inclure n'importe quelle procédure 'C' dans un programme en assembleur à condition de spécifier au moment de l'édition des liens

(LINKER) la librairie ‘C’ qui contient la procédure concernée. Les librairies du langage ‘C’ sont **CS.lib ou CM.lib ou CL.lib ou CH.lib.** Dans le cas d’un model mémoire SMALL la librairie est CS.lib. Prenons un Exemple :

Soit le programme suivant “**prog_asm.asm**” qui appelle les procédures ‘C’ suivantes :

- **int Cdecl printf(const char *__format, ...)**

- **char pascal LECTURE(void)**
- **void pascal AFFICHAGE (char x)**

Les procédures **LECTURE** et **AFFICHAGE** sont des procédures utilisateur et la procédure **printf** et une procédure ‘C’.

```
pile segment stack 'pile'
tab db 256dup(?)

pile ends

donnees segment para public 'donnees'
format byte 'le resultat est %d  %d',13,0

donnees ends

code segment para public 'code'


extrn  AFFICHAGE:far          ; procédure AFFICHAGE à importer

extrn  LECTURE:far            ; procédure LECTURE à importer

printf proto far C ,          ; procédure prototypé ‘C’ à importer
pstring:far ptr byte ,num1:word, num2:vararg
public _main



_main proc near               ; equivalent en C void main(void)

assume cs:code
assume ds:donnees
assume ss:pile
mov ax,donnees
mov ds,ax

push ds
pop es

mov bx,offset format
```

```
INVOKE printf, ds::bx,ax,di      ; appel de printf avec 4 paramètres
                                  ; adresse sur 32 bits
                                  ; registres ax et di à afficher par printf %i, %i
mov cx,10                        ; 10 caractères à lire et à afficher
boucle:    push cx                          ; passage du paramètre de la procédure
LECTURE
call  LECTURE                    ; procedure de lecture écrite en 'C'
                                        ; valeur de retour toujours dans AX
push ax                                 ; passage du paramètre de la procédure AFFICHAGE
call affichage                   ; affichage procédure d'affichge écrite en 'C'
pop cx                           ; boucle de lecture affichage de 10 caractères
loop boucle
mov ah,4ch                       ;revenir au SE
int 21h
_main endp
code ends
end _main
```

Le programme 'C' des procédures **char pascal LECTURE(void), void pascal AFFICHAGE (char x)** est le suivant ' prog_c.c ':

```
char pascal LECTURE(void)
{
char x;
x=getch();
return(x);
}

void pascal AFFICHAGE (char x)

{
putchar(x);
printf("%c",x);
}
```

Le programme prog_asm.asm doit être compilé bien évidement avec MASM :

c:\masm\bin\ > masm prog_asm.asm

Ceci permettra de générer un fichier objet **prog_asm.obj**. Le programme prog_c.c doit être compilé avec le compilateur 'C' :

c:\ tc\bin\> tcc /c prog_c.c ;

Ceci permettra de générer un fichier objet **proc_c.obj**

La troisième étape consiste à invoquer l'éditeur de liens :

c:\masm\bin\>**link c:\tc\lib\C0L.obj prog_asm.obj proc_c.obj**

Run File [prog.exe]:

List File [NUL.MAP]:

Libraries [.LIB]: **c:\tc\lib\CH.lib**

Definitions File [NUL.DEF] :

Le fichier prog.exe est alors généré et peut être exécuté:

c:\masm\bin\>prog ↵

I. Introduction.

Comme nous l'avions dis précédemment, un microcontrôleur est un composant réunissant sur un seul et même silicium un microprocesseur, divers dispositifs d'entrées/sorties et de contrôle d'interruptions ainsi que de la mémoire, notamment pour stocker le programme d'application. Il intègre également un certain nombre de périphériques spécifiques des domaines ciblés (bus série, interface parallèle, convertisseur analogique numérique, ...). Les microcontrôleurs améliorent donc l'intégration et le coût (lié à la conception et à la réalisation) d'un système à base de microprocesseur en rassemblant les éléments essentiels d'un tel système dans un seul circuit intégré. On parle alors de "système sur une puce" (en anglais : "System On chip"). Il existe plusieurs familles de microcontrôleurs, se différenciant par la vitesse de leur processeur et par le nombre de périphériques qui les composent. Toutes ces familles ont un point commun c'est de réunir tous les éléments essentiels d'une structure à base de microprocesseur sur une même puce. Voici généralement ce que l'on trouve à l'intérieur d'un microcontrôleur:

☞ Un processeur (C.P.U.),

☞ Des bus,

☞ De la mémoire de donnée (RAM et EEPROM),

☞ De la mémoire programme (ROM, OTPROM, UVPROM ou EEPROM),

☞ Des interfaces parallèles pour la connexion des entrées / sorties,

☞ Des interfaces séries (synchrone ou asynchrone) pour le dialogue avec d'autres unités,

☞ Des timers pour générer ou mesurer des signaux avec une grande précision temporelle.

Le microcontrôleur apparaît donc comme un système extrêmement complet et performant, capable d'accomplir une ou plusieurs tâches très spécifiques, pour lesquelles il a été programmé. Ces tâches peuvent être très diverses, si bien qu'on trouve aujourd'hui des microcontrôleurs presque partout: dans les appareils électroménagers (réfrigérateurs, fours à micro-ondes...), les téléviseurs et

magnétoscopes, les téléphones sans fil, les périphériques informatiques (imprimantes, scanners...), les voitures (airbags, climatisation, ordinateur de bord, alarme...), les avions et vaisseaux spatiaux, les appareils de mesure ou de contrôle des processus industriels, ... La force du microcontrôleur, qui lui a permis de s'imposer de manière si envahissante en si peu de temps, c'est sa spécialisation, sa très grande fiabilité et son coût assez faible (pour les modèles produits en grande série, notamment pour l'industrie automobile).

L'objectif premier est d'offrir le plus de performances et de services pour un prix minimal de la puce. Or aujourd'hui un coeur de processeur de 16 ou 32 bits représente une augmentation de la surface occupée sur le silicium de seulement quelques pour-cent par rapport à un circuit en 8 bits. Opter pour un coeur de processeur en 16 ou en 32 bits, c'est permettre entre autres des exécutions parallèles, un espace mémoire élargi, des interfaces de communications ou encore le remplacement de fonctions analogiques par des traitements numériques. Actuellement, une version 16 bits, voire 32 bits, a un coût comparable à un 8 bits. Les modèles 8 bits ont encore leur lot d'applications mais, trop souvent poussés à leur limite sans oublier l'espace mémoire qui reste inexorablement plafonné à 64 Ko.

II. Le processeur.

II.1 Structure classique.

Il est clair que la puissance d'un microcontrôleur est directement liée au processeur qu'il intègre. Ce processeur est surtout caractérisé par la famille à laquelle il appartient (CISC, RISC, VLIW, DSP). Il est constitué par un certain nombre d'éléments similaires à ce que l'on trouve dans un microprocesseur. Voici ce qu'on trouve à l'intérieur :

- Une Unité Arithmétique et Logique (UAL, en anglais *Aritmetic and Logical Unit - ALU*), qui prend en charge les calculs arithmétiques élémentaires et les tests.

- Une Unité de Contrôle.

- Des registres, qui sont des mémoires de petite taille (quelques octets), suffisamment rapides pour que l'UAL puisse manipuler leur contenu à chaque cycle de l'horloge. Un certains nombre de registres sont communs à la plupart des processeurs :

- ***Compteur d'instructions*** : Ce registre contient l'adresse mémoire de l'instruction en cours d'exécution.
- ***Accumulateur*** : Ce registre est utilisé pour stocker les données en cours de traitement par l'UAL.
- ***Registre d'adresses*** : Il contient toujours l'adresse de la prochaine information à lire par l'UAL, soit la suite de l'instruction en cours, soit la prochaine instruction.
- ***Registre d'instructions*** : Il contient l'instruction en cours de traitement.
- ***Registre d'état*** : Il sert à stocker le contexte du processeur, ce qui veut dire que les différents bits de ce registre sont des drapeaux (*flags*) servant à stocker des informations concernant le résultat de la dernière instruction exécutée.
- ***Pointeurs de pile*** : Ce type de registre, dont le nombre varie en fonction du type de processeur, contient l'adresse du sommet de la pile (ou des piles).
- ***Registres généraux*** : Ces registres sont disponibles pour les calculs.

- Un séquenceur, qui permet de synchroniser les différents éléments du processeur. En particulier, il initialise les registres lors du démarrage de la machine et il gère les interruptions.

- Une horloge qui synchronise toutes les actions de l'unité centrale. Elle est présente dans les processeurs synchrones, et absente des processeurs asynchrones et des processeurs autosynchrones

- Une unité d'entrée-sortie, qui prend en charge la communication avec la mémoire de l'ordinateur ou la transmission des ordres destinés à piloter ses processeurs spécialisés, permettant au processeur d'accéder aux périphériques de l'ordinateur.

II.2 Structures Actuelles.

Les processeurs actuels intègrent des éléments plus complexes :

- Plusieurs UAL, ce qui permet de traiter plusieurs instructions en même temps. L'**architecture superscalaire**, en particulier, permet de disposer des UAL en parallèle, chaque UAL pouvant exécuter une instruction indépendamment de l'autre.

- L'architecture superpipeline permet de découper temporellement les traitements à effectuer. C'est une technique qui vient du monde des supercalculateurs.

- Une unité de prédiction de saut, qui permet au processeur d'anticiper un saut dans le déroulement d'un programme, permettant d'éviter d'attendre la valeur définitive d'adresse du saut. Cela permet de mieux remplir le pipeline.

- Une unité de calcul en virgule flottante (en anglais *Floating Point Unit - FPU*), qui permet d'accélérer les calculs sur des nombres réels codés en virgule flottante.

- La mémoire cache, qui permet d'accélérer les traitements, en diminuant les accès à la RAM. Ces mémoires tampons sont en effet beaucoup plus rapides que la RAM et ralentissent moins le CPU. Le cache instructions reçoit les prochaines instructions à exécuter, le cache données manipule les données. Parfois, un seul cache unifié est utilisé pour le code et les données. Plusieurs niveaux de caches peuvent coexister, on les désigne souvent sous les noms de L1, L2 ou L3. Dans les microprocesseurs évolués, des unités spéciales du processeur sont dévolues à la recherche, par des moyens statistiques et/ou prédictifs, des prochains accès en mémoire centrale.

II.3 Jeu d'instructions.

On peut classer les instructions qu'un microprocesseur ou microcontrôleur est capable d'effectuer en quelques groupes.

II.3.1 Instructions de transfert.

Le microprocesseur ou microcontrôleur passe une grande partie de son temps à transférer des octets d'un emplacement vers un autre : d'un périphérique vers un registre interne, d'un registre interne vers la mémoire RAM ou vice-versa, d'un registre interne vers un périphérique ; ce qui ne peut en général pas être fait, c'est un transfert direct d'une case mémoire à une autre ou vers un périphérique; Quoique certains instructions de transfert utilisent comme opérande source un emplacement mémoire et comme opérande destination un autre emplacement mémoire. Ceci ne veut pas dire qu'un transfert direct mémoire-mémoire est possible. En faite l'information source est d'abord lue par le processeur, ensuite elle est écrite dans l'emplacement mémoire de destination.

II.3.2 Instructions arithmétiques

La majorité des microcontrôleurs ne comprennent que les instructions arithmétiques de base! Tout au plus ils sont capables d'effectuer des additions, des soustractions, des multiplications et des divisions sur des nombres binaires de 8 bits ou 16 bits. Toutes les opérations mathématiques complexes faisant intervenir des puissances, des racines carrées, des fonctions

trigonométriques, logarithmiques et exponentielles doivent être ramenées à une succession d'opérations simples portant seulement sur des octets. Des librairies mathématiques ont été développées pour la plupart des microcontrôleurs populaires pour faire, faire au microcontrôleur, des opérations arithmétiques complexes à partir des opérations arithmétiques de base (division, multiplication, soustraction, addition).

II.3.3 Instructions logiques

Les instructions logiques sont considérées comme étant les instructions les plus simples à faire du point de vue temps de calcul. Les opérations logiques de bases sont: ET, OU, XOU (*XOR*), NON (inverseur), rotations, décalages. La majorité des microcontrôleurs sont capables d'effectuer ces opérations (ET, OU, XOR, NON) sur des bits ce qui n'est pas le cas des microprocesseurs qui manipulent des informations multiples d'octets (8 bits, 16 bits, 32 bits, 64 bits voire 128 bits). La comparaison de deux octets A et B est considérée aussi comme une opération logique. Elle est réalisée comme une soustraction dont on néglige le résultat ; on s'intéresse simplement au résultat de la comparaison (A = B), ou (A > B) ou (A < B). Ces indications sont inscrites dans des indicateurs d'états (petites mémoires d'1 bit situées dans le registre d'état du processeur).

II.3.4 Instructions d'entrées/sorties

Ces instructions sont utilisées pour :

- lire l'état d'un port d'entrée (permettant l'interfaçage d'interrupteurs, de commutateurs, d'optocoupleurs, de convertisseurs analogiques/numériques, de claviers...) ;

- écrire une information dans le registre d'un port de sortie, qui maintient l'information à la disposition des circuits extérieurs : leds, moteurs, relais, convertisseurs numériques/analogiques... ;

- écrire ou lire une information dans les registres d'un port série.

Signalons que, dans certains microcontrôleurs, les périphériques sont considérés simplement comme des cases mémoire, et gérés par les instructions de transfert (**E/S intégrées mémoire**). D'autres microcontrôleurs disposent d'instructions spécifiques pour les E/S (**E/S indépendantes**).

II.3.5 Instructions de saut et de branchement.

Il s'agit d'instructions qui altèrent le déroulement normal du programme. On distingue les sauts et les branchements.

- Les **sauts** provoquent un branchement conditionnel ou inconditionnel du programme vers une adresse mémoire qui n'est pas contiguë à l'endroit où l'on se trouve. L'exécution du programme continuera à l'adresse du saut.

- Les branchements provoquent un saut vers un sous programme. Une fois l'exécution du sous programme faite le processeur pointe sur l'instruction qui est juste après l'instruction qui a provoqué le branchement vers le sous programme. La grande différence par rapport au saut, c'est qu'au moment du branchement il faut mémoriser l'adresse d'où l'on vient, afin de pouvoir y revenir une fois le sous-programme terminé. La mémorisation de l'adresse de départ se fait par l'intermédiaire d'un registre interne du processeur appelé souvent stack pointer ou pile.

Les saut et branchements peuvent être :

- inconditionnels ;

- conditionnels, c'est à dire que le branchement n'a lieu que si une certaine condition est remplie ; généralement le test se fait par rapport à l'un des bits du registre d'état ; ceux-ci indiquent par exemple si le contenu de l'accumulateur est nul, positif, négatif, de parité paire ou impaires.

II.3.6 Instructions diverses.

On trouve dans ce groupe :

- des instructions de gestion de la pile. La pile est une zone de mémoire RAM gérée automatiquement par le microcontrôleur pour la sauvegarde des registres ou pour la mémorisation des adresses de retour en cas d'un branchement vers un sous programme.

- des instructions de contrôle du processeur : par exemple passage en mode basse consommation, contrôle des périphériques embarqués (c'est à dire sur la même puce que le processeur) ;

- des instructions permettant de positionner des indicateurs internes du processeur.

II.4 Modes d'adressage pour les données.

De nombreuses instructions font référence à des données se trouvant à différents endroits du microcontrôleurs : registres internes du processeur, RAM, EEPROM, ports d'E/S,

périphériques intégrés. On appelle modes d'adressage les différentes façons de spécifier les endroits où se trouvent les données dont on a besoin.

II.4.1 Adressage implicite.

Certaines opérations ne peuvent être réalisées que sur une donnée se trouvant en un endroit bien précis du processeur (par exemple, l'accumulateur ou la pile). Dans ce cas, il n'est pas nécessaire de spécifier l'adresse du registre en question et on parle d'**adressage implicite**.

II.4.2 Adressage registre ou inhérent.

Le processeur dispose d'un certain nombre de registres de travail. De nombreuses instructions y font référence ; vu leur nombre peu élevé (8, par exemple), il suffit d'un petit nombre de bits pour spécifier le registre désiré (3 bits dans notre cas). On parle dans ce cas d'**adressage registre** ou **inhérent**.

II.4.3 Adressage direct.

Dans ce mode d'adressage, on donne l'adresse (généralement en 16 bits) de la donnée en mémoire (RAM, ou port d'E/S s'il est intégré à la mémoire). Ce mode d'adressage permet d'indiquer n'importe quel endroit dans la mémoire, l'inconvénient étant que l'on doit spécifier l'adresse concernée dans son intégralité (2 à 4 octets). Ce qui conduit à des instructions assez longues. Certains processeurs implémentent aussi, pour réduire l'encombrement du programme, l'adressage direct restreint : l'adresse ne comporte qu'un octet, et ce mode ne permet d'accéder qu'aux données se trouvant sur la page courante. Un registre de sélection de page est généralement utilisé pour spécifier la page mémoire en cours.

II.4.4 Adressage indirect à registre.

Dans ce mode d'adressage, l'adresse de la donnée se trouve dans un registre spécial du processeur (du même nombre de bits que son bus d'adresses), le pointeur de données. L'avantage, par rapport à l'adressage direct, est que l'adresse peut être manipulée commodément, par exemple pour accéder à une suite de données consécutives en mémoire. Ceci est particulièrement utile lorsqu'on manipule des données stockées dans un tableau.

II.4.5 Adressage immédiat.

C'est un peu un abus de langage que de parler d'adressage dans ce cas-ci. En effet, la donnée suit tout simplement l'instruction.

II.4.6 Adressage indexé.

Ce mode est assez semblable à l'adressage indirect à registre. Il fait appel à un registre spécial appelé **registre d'index**. Certains microprocesseurs ou microcontrôleurs ne supportent pas ce mode, d'autres au contraire ont 1 ou même 2 registres d'index. Deux registres d'index sont

particulièrement bienvenus lorsqu'il s'agit de déplacer un bloc de données dans la mémoire RAM d'un emplacement vers un autre. Le premier servira alors pour pointer la zone mémoire source et le second pour pointer la zone mémoire destination.

III. Le choix d'un microcontrôleur.

Nous n'avons pour l'instant évoqué que des généralités applicables à tous les microcontrôleurs du marché, sans citer de marque précise. En effet, toute la difficulté du choix d'un microcontrôleur pour une application donnée réside dans la sélection du "bon" circuit adapté pour cette application. Le choix du microcontrôleur est surtout dicté par deux critères principaux :

- l'adaptation de son architecture interne aux besoins de l'application (présence de convertisseurs A/N par exemple ou d'un timer disposant d'un mode particulier, ...);

- le fait de posséder déjà ou non un système de développement.

En effet, si l'on ne possède rien, on peut se laisser guider par le premier critère en comparant toutefois les investissements de développement à prévoir. Si l'on est déjà équipé, mieux vaut choisir un circuit un peu moins bien adapté, quitte à lui adjoindre des circuits externes, que le circuit qui va bien mais qui impose un changement de système. Pour simplifier un peu ce deuxième dilemme, les fabricants ont essayé de développer non pas des microcontrôleurs isolés mais des familles de circuits, plus ou moins compatibles entre eux tant au niveau de l'architecture qu'au niveau de la programmation et des outils de développement. Il existe plusieurs familles de microcontrôleurs dont les plus connues sont :

- la famille Atmel AT91
- la famille Atmel AVR
- le C167 de Siemens/Infineon
- la famille Hitachi H8
- la famille Intel 8051, qui ne cesse de grandir ; de plus, certains processeurs récents utilisent un cœur 8051, qui est complété par divers périphériques (ports d'E/S, compteurs/temporisateurs, convertisseurs A/N et N/A, chien de garde, superviseur de tension...)
- l'Intel 8085, à l'origine conçu pour être un microprocesseur, a en pratique souvent été utilisé en tant que microcontrôleur

- le Motorola 68HC11
- la famille des PIC de Microchip
- la famille des ST6 de STMicroelectronics
- la famille ADuC d'Analog Devices
- la famille PICBASIC de Comfile Technology

Il est bien évident que, dans le cadre de ce cours dont le nombre de pages doit forcément rester limité, il ne va pas être possible de donner toutes les informations, matérielles et logicielles, relatives à tous ces microcontrôleurs. Le manuel technique de chacun d'entre eux comporte en effet plusieurs dizaines de pages, voir parfois une centaine. Nous allons nous intéresser dans le cadre de ce cours à la famille Intel C51. Cette famille, qui ne cesse de s'agrandir, est basé sur l'architecture 8051 qui est intemporelle et continuera d'évoluer, car de nombreux concepteurs ne souhaitent pas renoncer à l'investissement réalisé dans les progiciels, les outils et l'expertise acquise avec le 8051. Il est vrai que l'utilisation du langage C, d'interfaces utilisateur plus conviviales mais plus lourdes, de jeux de caractères multiples, de vitesses de transmission accélérées et l'enregistrement des données de masse nécessitent des densités de mémoires beaucoup plus importantes que celle proposée avec la majorité des microcontrôleurs de cette famille. Toutefois si nous prenons le cas de la famille µPSD3200 basé sur l'architecture 8051, elle possède deux bancs indépendants de mémoire Flash (256 ko et 32 ko), capables de fonctionner simultanément en lecture et en écriture, 8 ko de SRAM et plus de 3000 portes de logique programmable avec 16 macrocellules. Le jeu de périphériques intègre une interface USB, deux canaux UART, quatre unités PWM 8 bits, quatre canaux ADC 8 bits, une interface maître-esclave I²C, un canal d'affichage de données, des fonctions de supervision comme l'horloge chien de garde et la détection de basse tension, et jusqu'à 50 broches d'entrée/sortie multi-usages. Il est clair que ce modèle de microcontrôleur 8 bits basé sur l'architecture C51 n'a rien à envier aux modèles 16 et 32 bits. Il convient particulièrement aux systèmes embarqués nécessitant de grandes quantités de stockage de code et/ou de données, à l'instar des périphériques utilisés sur les lieux de vente : lecteurs de chèques et de cartes, imprimantes thermiques, lecteurs de codes-barres et contrôleurs de distributeurs automatiques. Autres applications également bien prises en charge : la sécurité des immeubles, les alarmes, le contrôle d'accès, le contrôle industriel, les applications GPS portables, les téléphones publics et l'instrumentation. Il est bien clair d'après cet

exemple de la famille µPSD3200 basée sur l'architecture 8051, que les microcontrôleurs 8 bits ont encore leur lot d'applications et qu'ils continueront à concurrencer les microcontrôleurs 16 et 32 bits.

La famille µPSD3200 de St Microelectronics n'est qu'un exemple, en effet plusieurs constructeurs font aujourd'hui des microcontrôleurs utilisant l'architecture 8051.

IV. La Famille MCS51

La famille MCS-51 est l'une des plus prolifiques parmi celles des microcontrôleurs. Ce processeur créé tout au début des années quatre-vingt par la société INTEL a connu au fil des années de nombreux descendants. On distingue actuellement de très nombreuses variantes fournies par six principaux constructeurs (Intel, Dallas-Semiconductor, Philips/Signetics, Oki, Siemens, Amd, Matra-Harris), sans compter quelques modèles peu répandus, réalisés par des fabricants moins connus. Cette famille MCS-51 est la plus vendue (ou la seconde selon les sources) sur le marché mondial des microcontrôleurs 8 bits non dédiés. Le marché des microcontrôleurs, qui était le second de l'électronique, est maintenant le premier devant celui des mémoires, loin devant celui des microprocesseurs (source Electronic World News). Les circuits MCS-51 sont fabriqués selon les deux technologies MOS (NMOS et CMOS) et plusieurs de leurs variantes. Alors que le composant NMOS classique consomme environ de 100 mA à 300 mA (selon la variante) sous une tension de 5V, certains types en technologie CMOS peuvent se contenter de 1,8V sous 2,5 mA (on trouve ces modèles dans les cartes à puce et les téléphones auto alimentés). Le microcontrôleur de base, qui incorpore un certain nombre de fonctions périphériques élémentaires, est commercialisé en un boîtier de 40 broches (DIL) ou de 44 broches (boîtier plat PLCC). Les variantes, épurées ou alourdies de nombreux périphériques supplémentaires, sont incluses dans des boîtiers de 24 broches (boîtier DIL étroit) à 84 broches (boîtier carrés plats divers). Le composant devenant ancien, la société Intel a (en raison du succès exceptionnel du contrôleur) mis sur le marché une version modernisée compatible appelée 80C251 et comportant un pipeline d'exécution rendant le processeur quatre fois plus rapide à fréquence d'horloge égale. D'autres améliorations comme le fonctionnement statique, une consommation nettement moindre, une capacité d'adressage étendue ont été apportées. Philips, pour sa part, commercialise depuis peu une variante 16 bits

appelée 80C51-XA compatible au niveau du code source aux performances également fortement améliorées par rapport au composant initial.

Les différentes versions possèdent en outre quelques caractéristiques spécifiques supplémentaires. Ainsi le 8052AH possède 6 entrées d'interruption au lieu de 5 dans la version de base, les versions HCMOS possèdent des possibilités de mise en veille pour réduire la consommation, le 83C152 dispose de deux canaux d'accès direct à la mémoire, ainsi que d'un cinquième port et possède un brochage différent. De plus une version spéciale du 8052 dispose du basic intégré en rom (8052AH-BASIC). Le tableau ci-dessous présente les principales versions construites par INTEL.

nom	version romless	version eprom	tailles rom en bytes	bytes ram	timers 16 bits	technologie
8051AH	8031AH	8751H	4K	128	2	HMOS (2)
8052AH	8032AH	8752BH	8K	256	3	HMOS (2)
80C51BH	80C31BH	87C51	4K	128	2	HCMOS
83C51FA	80C51FA	87C51FA	8K	256	4	HCMOS
83C152	80C152	87C152	8K	256	2	HCMOS

Tableau 4.1. Famille MCS51

Dans un premier temps nous allons examiner la version de base à savoir le 80C51BH, puis par la suite nous allons nous intéresser au microcontrôleur 80C552. Ce microcontrôleur de PHILIPS, basée sur l'architecture 8051, dispose d'une RAM interne plus grande, de ports d'E/S plus nombreux, de convertisseurs A/N, de C/T supplémentaires (9 au total), d'une interface I2C maître esclave.

V. Le microcontrôleur 80C51

Les performances du 8051 comparées à celles de microprocesseurs 16 bits classiques permettent d'avoir une idée des possibilités du composant. Dans le cas d'une application de contrôle, typique de ce que l'on peut demander à un microcontrôleur, la vitesse de calcul d'un 8051 à 12 MHz est comparable à celle d'un 8086 à 8MHz (compatible PC XT TURBO), ou d'un 68000 (MACINTOSH Classique). Les versions à haute vitesse (horloge à 30 MHz et plus) du 8051 donnent des résultats comparables à ceux des 80286 à 8 MHz (compatibles AT TURBO), ou 68020 à cadencement moyen. Le rapport de performance dépend considérablement du type de logiciel exécuté. Pour les calculs booléens, le 8051 triple sa performance relative par rapport aux autres processeurs (grâce à son processeur booléen) ; par

contre, dans le cas de calculs arithmétiques sur des entiers de 16 ou 32 bits, sa performance s'écroule. Le noyau du 80C51 reprend partiellement les principes des processeurs RISC (décodeur d'instructions entièrement câblé et jeu d'instructions réduit). Il exécute 70% des instructions en un cycle machine, et les autres en deux cycles (hormis celles de multiplication et de division qui s'exécutent en quatre cycles). A titre de comparaison, un 8086 qui est microprogrammé demande plus de 20 cycles machine pour exécuter certaines instructions de chargement. L'unité centrale du 80C51 incorpore un processeur booléen qui accroît considérablement la vitesse de traitement des instructions de manipulation de bits. Le microcontrôleur 8051 est un microcontrôleur 8 bits car il traite des données sur 8 bits ; le bus de données comporte donc 8 lignes ; comme la plupart des microcontrôleurs 8 bits, le 8051 gère des adresses en 16 bits, ce qui donne un espace adressable de 2^{16} soit 64Koctets, où K = 2^{10} = 1024. Il exploite l'architecture de Harvard, plus complexe mais plus performante. Dans cette architecture, la mémoire de données est physiquement distincte de la mémoire de programme, chacune de ces deux mémoires ayant son propre dispositif d'adressage et de contrôle. La mémoire interne de données est une mémoire vive volatile (de 128 octets à 512 octets selon les modèles), donc très réduite; tandis que la mémoire interne de programme est une mémoire morte (reprogrammable ou non) dont la taille varie de 4 Koctets à 32 Koctets selon le modèle de contrôleur. Il est possible de rajouter de la mémoire à l'extérieur du contrôleur. La limite maximale est de 64 Koctets de mémoire de données et autant de mémoire de programme. Le processeur ne gère pas le rafraîchissement des mémoires dynamiques, ni les temps d'attente des mémoires lentes (simplicité oblige). Il est donc nécessaire de choisir des mémoires statiques ayant des temps d'accès suffisamment courts. Les caractéristiques principales du noyau 80C51 sont :

- un CPU à 8 bits spécialement conçu pour la commande d'applications diverses
- une unité logique étendue au traitement sur un bit
- 32 entrées/sorties bidirectionnelles qui peuvent être adressées individuellement.
- 128 octets de RAM interne à utilisation générale
- 21 registres spécialisés
- un port série en full duplex
- 5 sources d'interruptions avec 2 niveaux de priorité
- 2 Compteurs/Timers sur 16 bits
- un oscillateur interne: la fréquence d'oscillation maximale admise est de 12 MHz
- 64 Ko d'adresse mémoire de données
- 64 Ko d'adresse mémoire de programme
- un jeu d'instructions assez développé

Ce noyau de l'architecture de base est présenté par le schéma bloc de la figure suivante :

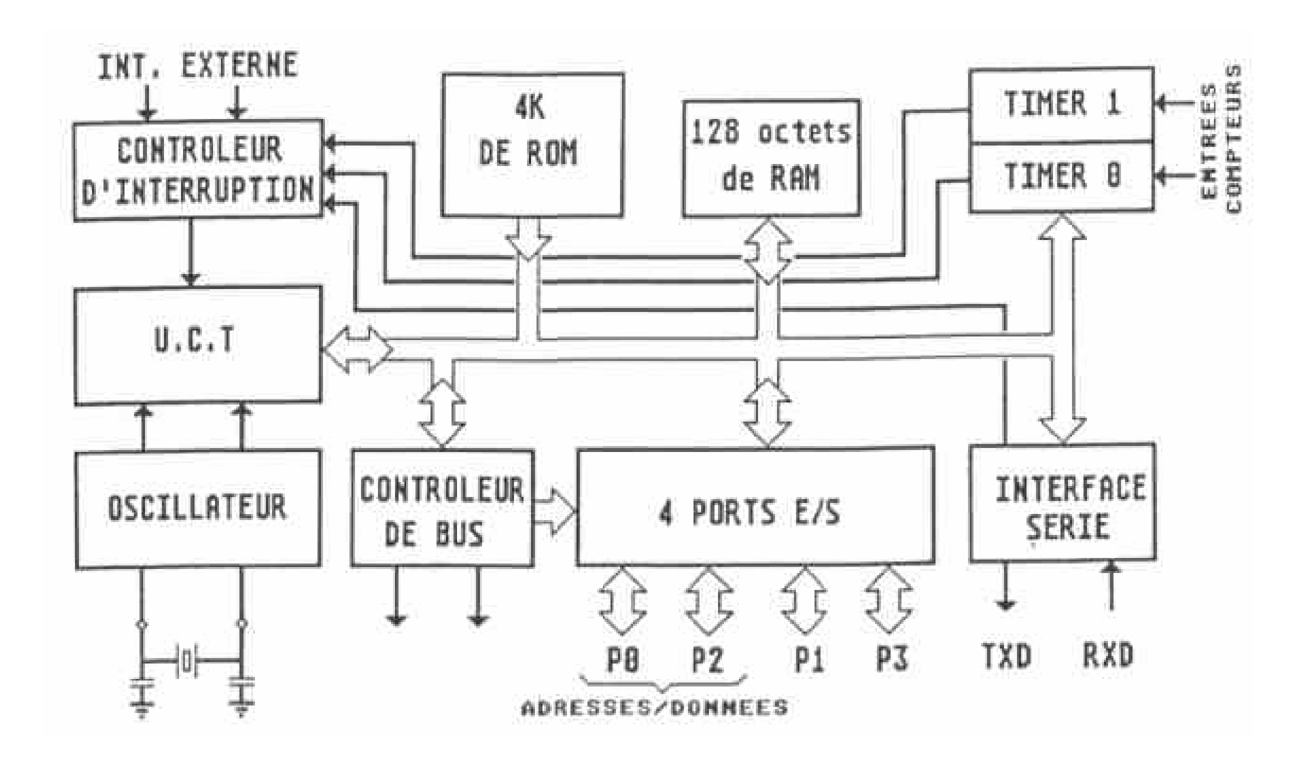


Figure 4.1. Schéma bloc du noyau 80C51

Tous les microcontrôleurs de la famille 8051 sont caractérisés par la particularité de pouvoir adresser d'une façon séparée un espace mémoire de programme et un espace mémoire de données. La mémoire de données comporte en fait deux zones d'une part les 128 (ou 256) octets internes au microcontrôleur et d'autre part la RAM externe. L'organisation des 128 octets internes est donnée par la figure suivante (Annexe N°1) :

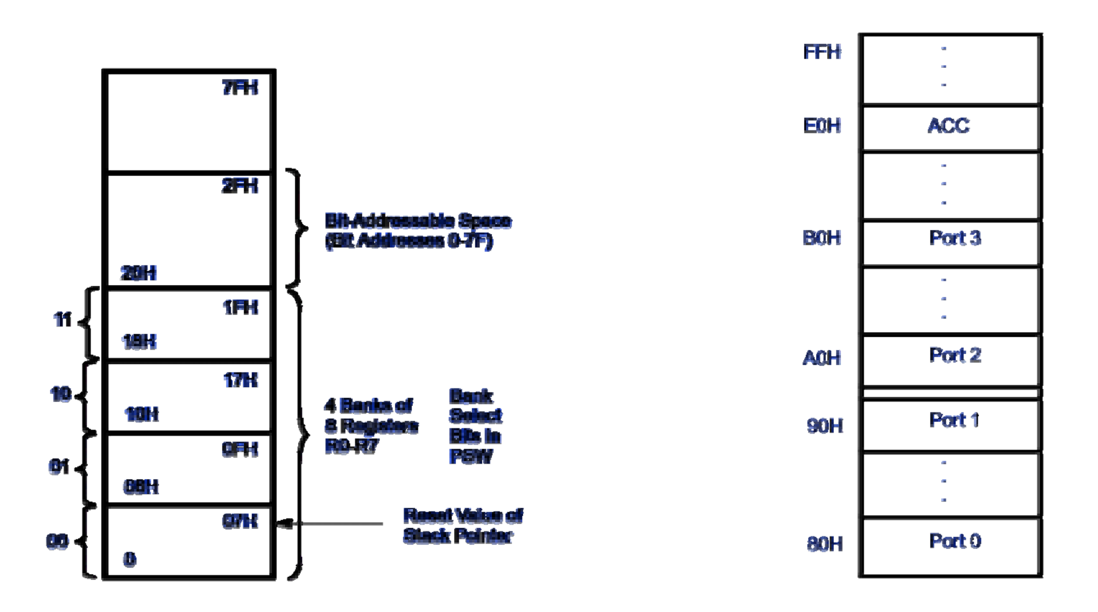


Figure 4.2. Organisation de la mémoire interne

En ce qui concerne la mémoire externe, la figure suivante résume un peu la situation. Nous avons 64 KO de mémoire programme et 64 KO de mémoire de données.

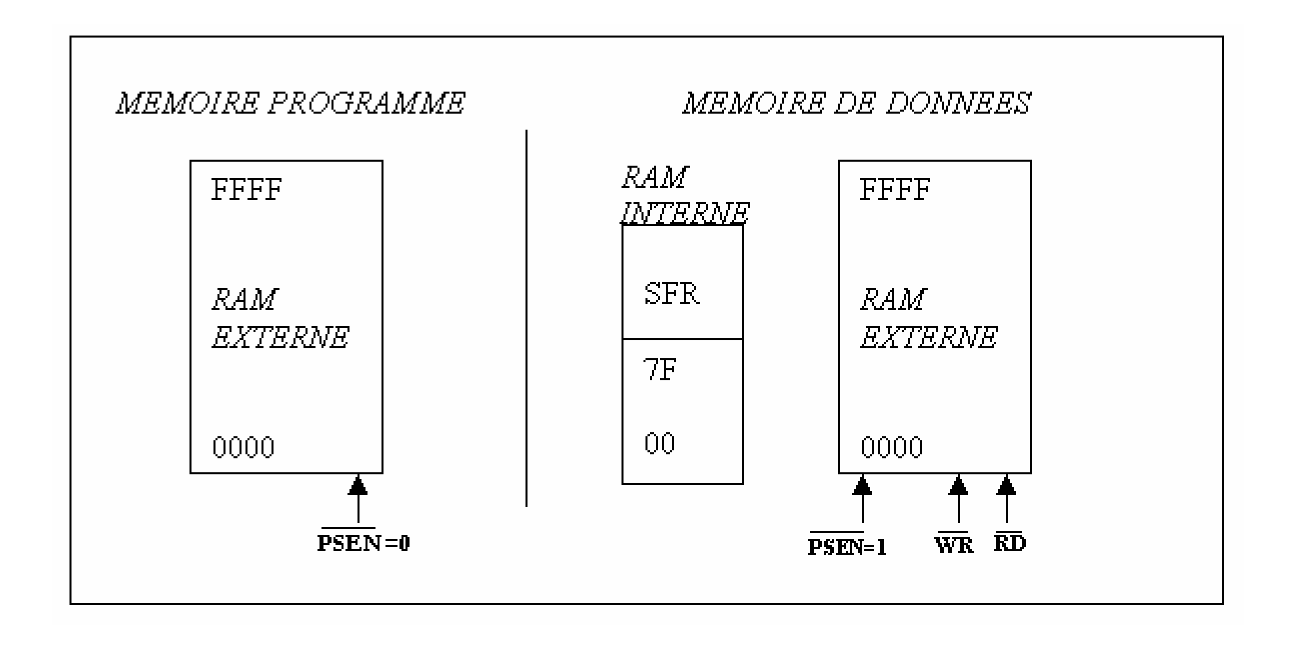


Figure 4.3. Organisation des espaces mémoires du 8051

V.1 Mémoire interne de données.

Cette mémoire se décompose en deux parties de 128 octets chacune. Le bloc de 128 octets situé de l'adresse 0 à l'adresse 7FH constitue la partie basse de cette mémoire interne et il est présent dans tous les microcontrôleurs de la famille MCS 51. Il en est de même pour le bloc nommé SFR. L'appellation SFR se traduit par Registres à Fonction Spéciale. Ces registres occupent les adresses directes 80H à 0FFH. Ces différents registres sont décrits en détail plus loin.

V.2 Séparation logique entre programme et données.

On désigne par mémoire de programme la mémoire dans laquelle se trouve le code machine exécuté par le processeur, par opposition à la mémoire de données où se trouvent les données manipulées par le processeur. Cette séparation de fonctions se traduit par des modes d'accès et d'adressages différents. La mémoire de données est accessible aussi bien en lecture qu'en écriture. Une RAM interne est prévue à cet effet. Si la capacité de celle-ci s'avérait insuffisante, il est possible de compléter l'espace de mémoire de données grâce à une mémoire externe. Le microcontrôleur peut commander cette mémoire en lecture et écriture à l'aide des signaux RD et WR.

V.3 Addition d'une mémoire externe au microcontrôleur.

Bien que la fonction première d'un microcontrôleur soit de permettre un contrôle et une commande d'un système limité et fini, il peut aussi se présenter comme une alternative à l'utilisation d'un microprocesseur. Il faut donc qu'il soit capable de gérer des plans mémoires externes ou des composants périphériques. Une solution consiste alors à doter certaines broches correspondantes à des lignes d'entrées/sorties, d'une deuxième fonction dite fonction secondaire. C'est le cas des broches du port P0 et P2 qui pourront devenir bus de données et d'adresses. Ils existent plusieurs configurations possibles de mémoires externes. La plus simple est donnée par la figure suivante :

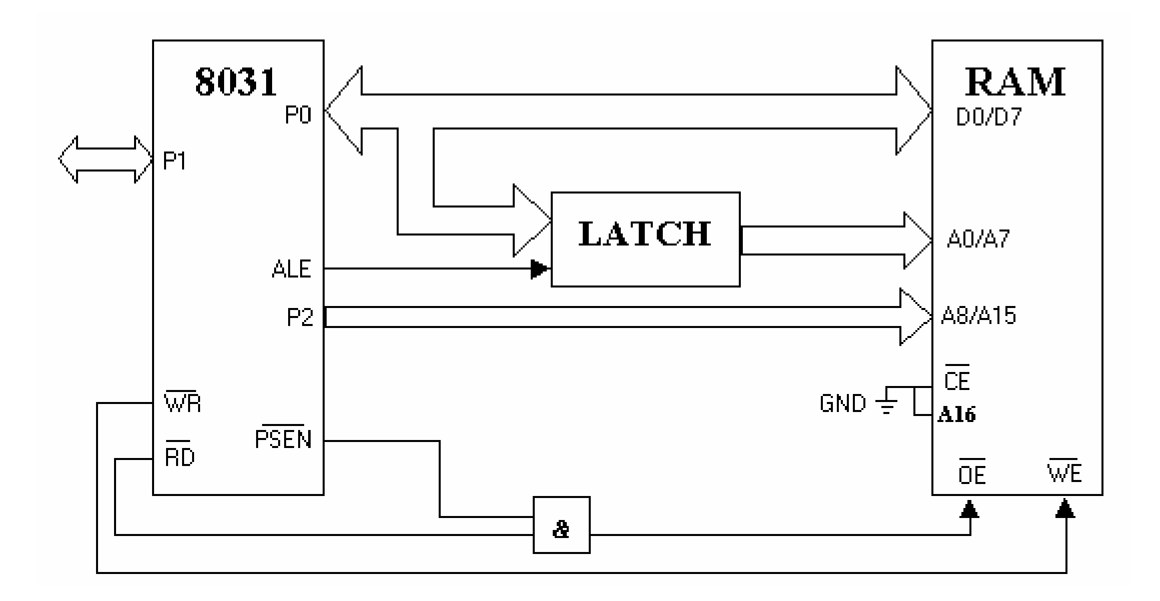


Figure 4.4. Association Microcontrôleur et mémoire externe

Sur cette figure la mémoire externe de type RAM est à la fois mémoire programme et mémoire de données. Bien évidemment, dans cette configuration, le programme d'application doit se trouver préalablement dans cette mémoire ce qui implique l'utilisation de mémoire RAM non volatiles ou sauvegardées par pile. La fonction du LATCH est de démultiplexer le bus d'adresse et de donnée tandis que le rôle de la fonction ET est de considérer la mémoire comme étant à la fois mémoire de programme et de données. Remarquons sur cette figure que le démultiplexage des données et des 8 adresses de poids faible est possible grâce à la propriété de la broche ALE (Adress Latch Enable). En faite cette dernière est prévue pour commander le démultiplexage du port P0 lorsque celui-ci est validé dans sa fonction secondaire : poids faible du bus de données.

Lorsque ALE est dans l'état 1, le port P0 présente la partie A0/A7 de l'adresse. Lors de la transition 1 vers 0 de ALE, l'adresse toujours présente doit être mémorisée grâce à un circuit externe 'LATCH' (bascule D). Durant la période où ALE=0, le port P0 devient bus de données. Les pattes CE et A16 de la RAM sont liées ici à la patte GND, ainsi la mémoire externe est toujours sélectionnée : La zone mémoire disponible dans ce cas est 0000H - FFFFH. Pour charger un programme dans la mémoire externe, il faut qu'elle soit accessible en tant que mémoire de données. Alors que dans la phase d'exécution de ce programme, il faut qu'elle soit reconnue par le microcontrôleur comme une mémoire programme. Au niveau logique cette fonction mixte est

assurée par un 'ET' logique entre les signaux PSEN et RD. La sortie de ce port logique est affectée à OE. En effet la sortie PSEN (Program Store Enable) passe à l'état logique 0 dès que le microcontrôleur entreprend la récupération d'une instruction de la mémoire programme externe. Il faut noter que lors d'un accès à la mémoire externe de données, cette sortie reste à l'état 1. Dans ce qui suit nous présentons la chronologie de ces signaux lors de la récupération d'une donnée et d'un code depuis la mémoire externe.

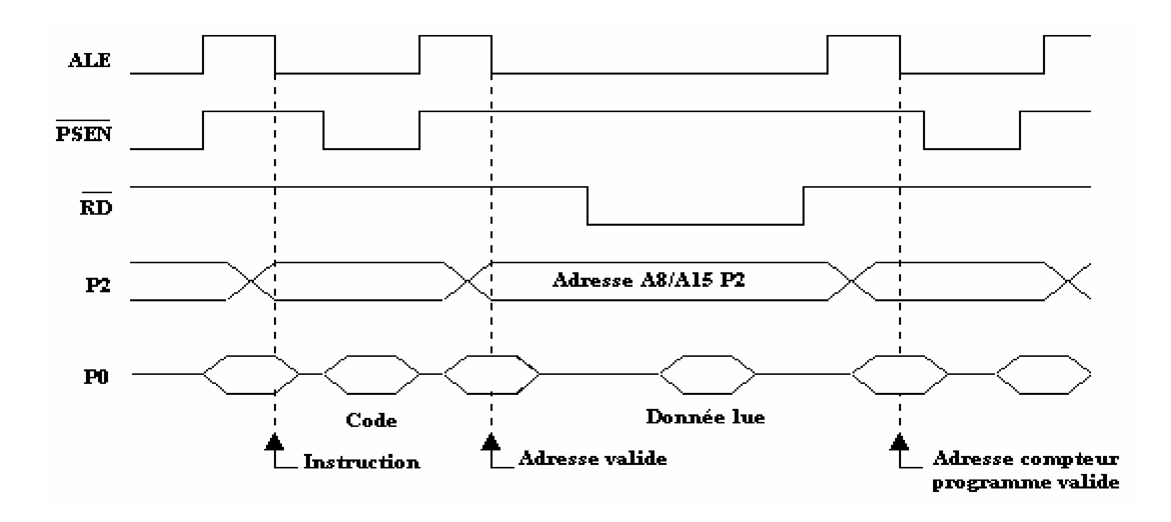


Figure 4.5. Chronogramme des signaux lors de la récupération d'une donnée et d'un code depuis la mémoire externe

Sur la figure suivante nous présentons la chronologie de ces signaux lors de l'écriture d'une donnée dans la mémoire externe.

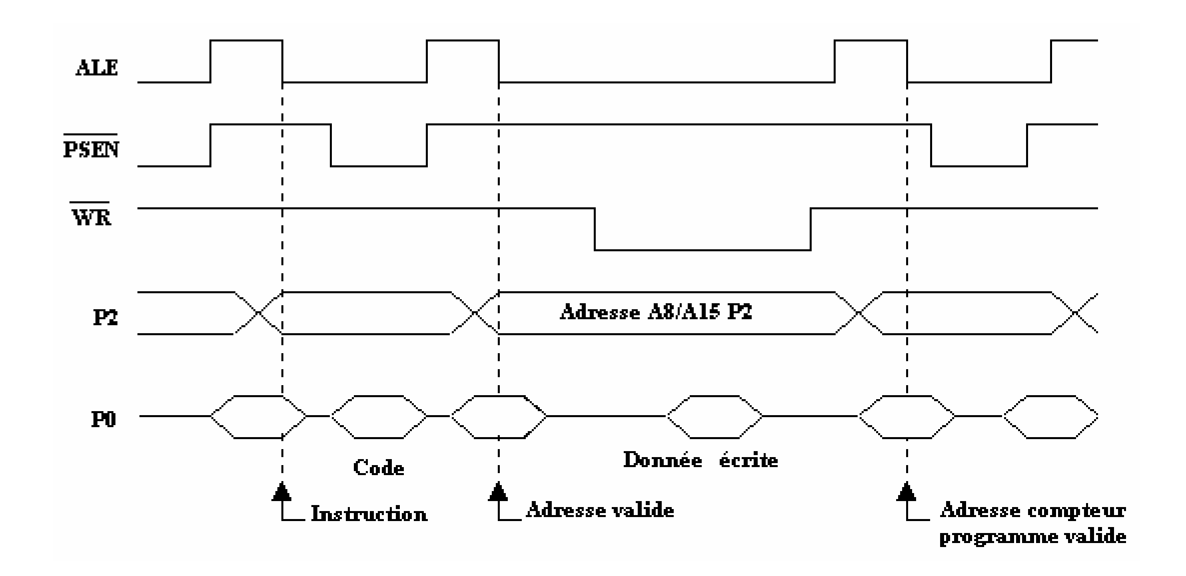


Figure 4.6. Chronogramme des signaux lors de l'écriture d'une donnée dans la mémoire externe

Pour confirmer expérimentalement ces chronogrammes nous donnons sur les figures 4.7 et 4.8 quelques résultats que nous avons visualisés sur l'oscilloscope lors de la lecture et l'écriture d'une donnée. Notons que nous visualisons ici seulement la patte D6 des données.



Figure 4.7. Chronogramme des signaux lors de la récupération d'une donnée depuis la mémoire externe

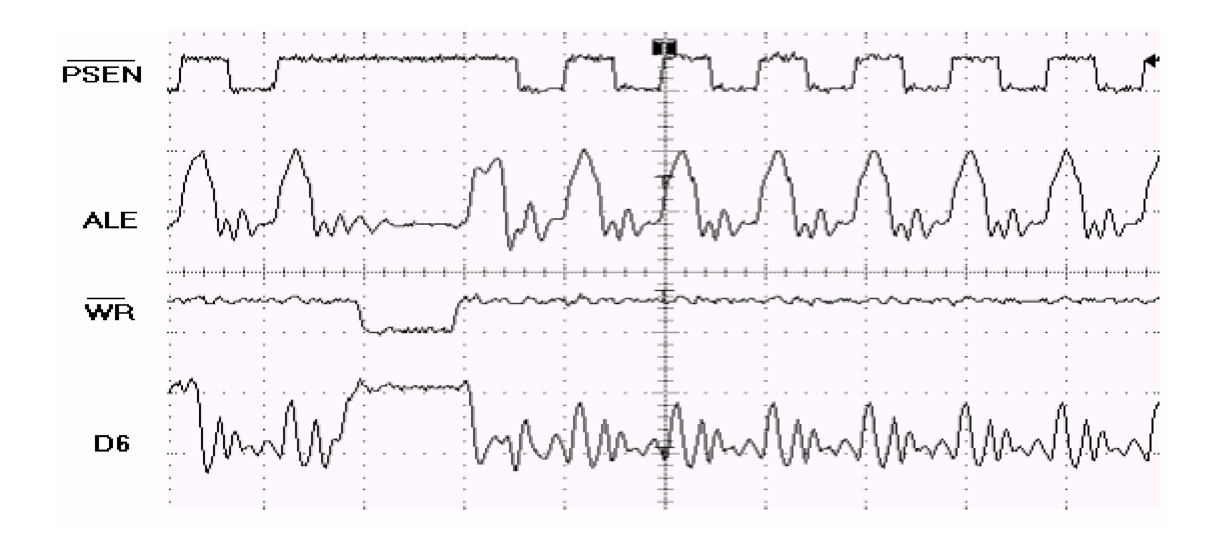


Figure 4.8. Chronogramme des signaux lors de l'écriture d'une donnée dans la mémoire externe

La deuxième configuration possible lors d’une extension mémoire pour un 80C51 et de séparer la mémoire du code de celle des données. Pour mieux comprendre cette séparation, voici un exemple de mise en oeuvre d'un 8032 associé à une EPROM de 8Ko pour la mémoire programme et à une RAM de 8Ko également pour stocker des données:

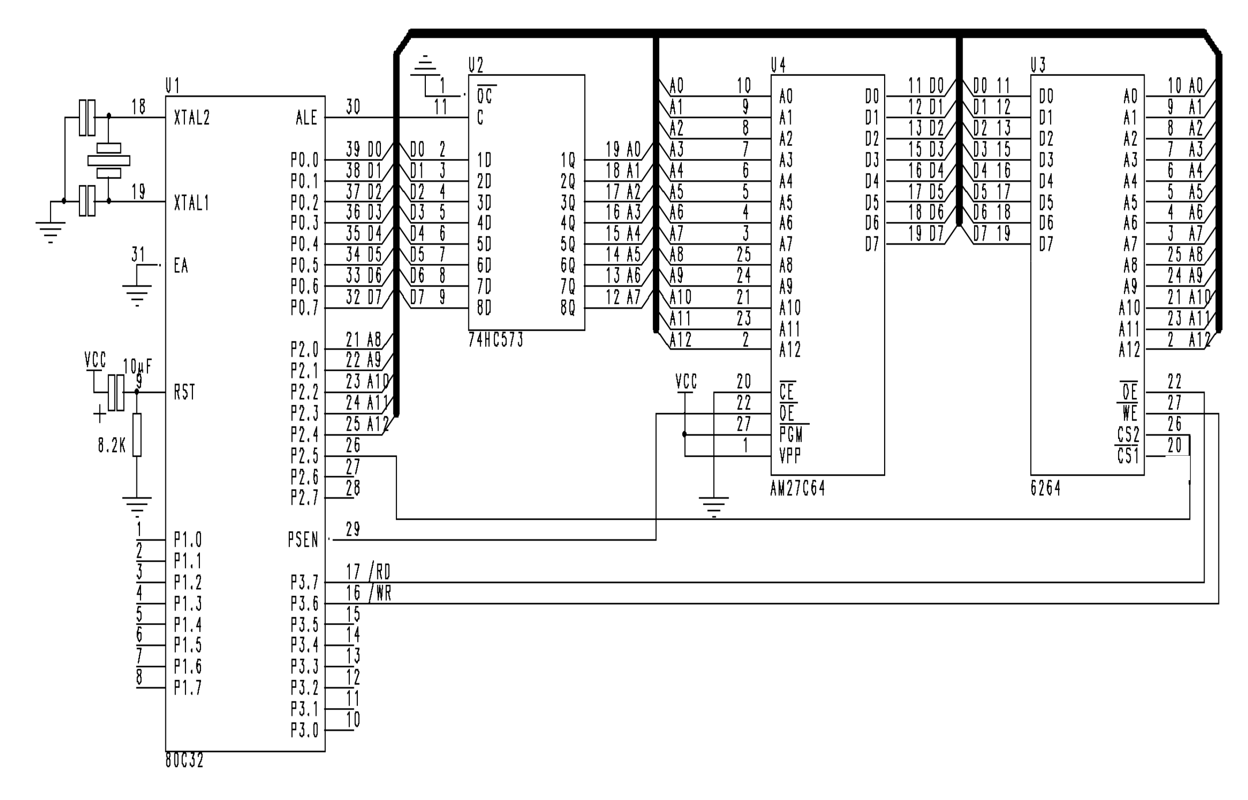


Figure 4.9. Mise en œuvre du microcontrôleur C51 avec séparation entre

mémoire de code et mémoire de données

Le 74HC573 est un verrou 3 états il permet le démultiplexage du port P0. Lorsque ALE est à 1, on recopie les entrées du latch (74HC573) sur les sorties et lorsque ALE passe à 0, les données restent sur les sorties du 74HC573, elles sont les adresses de A0 à A7 et le microcontrôleur se sert alors de son port P0 pour échanger des données. On constate ici que /PSEN commande directement le signal /OE de l'EPROM. Quand le microcontrôleur veut aller chercher des instructions dans sa mémoire programme, PSEN passe à 0 et l'EPROM est alors validée en lecture. La RAM est dévalidée car son CS est commandé par A13. Dans le cas où l'on désire avoir 64Ko de RAM, il faut prévoir un décodage d'adresse moins rudimentaire.

V.4 Structure du processeur C51.

Comme nous l'avions déjà vu au paragraphe II., Les parties essentielles d'un processeur sont : L'Unité Arithmétique et Logique, L'Unité de Contrôle, le Compteur d'instructions, l'Accumulateur, le Registre d'adresses, le Registre d'instructions , le Registre d'état, le Pointeurs de pile, les Registres généraux, le séquenceur et l'horloge. Le processeur C51 comme la majorité des processeurs intègre tous ces éléments sur une même puce. Il est chargé bien évidemment d'exécuter toutes les instructions du programme d'application. Cette exécution se résume par trois étapes : traitement de données, Gestion des adresses et traitement des instructions.

V.4.1 traitement des données.

Cette étape implique 3 registres et une **unité arithmétique et logique** (UAL). Les données manipulées par le 8051 sont soit des octets ou tout au plus des mots (16 bits). La majorité des opérations effectuées sur ces données utilise implicitement l'accumulateur comme registre de destination. Le principe de fonctionnement est le suivant: les registres de 8 bits TMP1 et TMP2 (registres à usage générale) présentent à l'UAL la ou les données à traiter. Après traitement, le résultat est placé dans l'accumulateur (Acc). Ce résultat peut être alors envoyé dans une mémoire (RAM ou registre de travail) ou vers un port de sortie.

L'UAL comporte les circuits suivants :

- un additionneur pour 2 nombres de 8 bits ; le 9ème bit de l'addition est placé dans l'indicateur d'état CY (Carry) ;

- un soustracteur pour 2 nombres de 8 bits ; l'emprunt éventuel généré si le résultat est négatif est aussi placé dans un indicateur d'état ;
- un circuit de multiplication pour 2 nombres de 8 bits ; ici, le résultat peut nécessiter jusqu'à 16 bits ; les 8 bits les plus significatifs sont placés dans un registre B prévu à cet effet ;
- un circuit de division pour 2 nombres de 8 bits ; il s'agit ici d'une division entière : le quotient est placé dans l'Acc, le reste dans le registre B ;
- 8 circuits logiques de type inverseur, ET, OU, XOU ;
- de nombreux tampons 3 états pour permettre les communications entre les registres, l'UAL et le bus de données.

Le registre d'état comprend un certain nombre d'**indicateurs d'état**, qui sont automatiquement positionnées lors des opérations arithmétiques ou logiques. L'état de ces indicateurs peut être testé par le programme d'application pour effectuer des branchements conditionnels. En plus de ce registre le 8051 comporte 32 autres registres de travail, organisés en 4 groupes de 8 registres. Ces registres constituent des mémoires temporaires, où l'on place provisoirement des informations dont on sait qu'elles seront nécessaires peu de temps après. L'intérêt des registres de travail, c'est que le temps d'accès est plus court que pour une lecture ou écriture en RAM, et les instructions nécessaires pour y accéder sont plus rapides.

V.4.2 Gestion des adresses.

L'espace mémoire d'un 8051 est partagé en 3 zones bien distinctes :

- la **zone mémoire programme** qui peut aller jusqu'à 64 Koctets;

- la **zone mémoire de données**, qui peut aussi aller jusqu'à 64 Koctets

- la **pile** : il s'agit d'une zone de mémoire incluse dans la zone mémoire de données et qui est utilisée par le processeur principalement pour stocker les adresses de retour au programme principal en cas d'appel de sous programmes; on peut également l'utiliser comme zone de stockage temporaire (un peu comme les registres de travail) ou pour l'échange de données entre programme principal et sous-programmes.

Le 8051 comporte donc très naturellement trois registres consacrés à la gestion des adresses.

- Le **pointeur de programme**, aussi appelé compteur de programme (PC, *Program Counter*), est un registre de 16 bits qui contient à tout moment l'adresse de la prochaine instruction à exécuter. Il est mis à 0 au moment de la mise sous tension du

système ; le programme doit donc impérativement commencer à l'adresse 0000h (h signifiant code hexadécimal, c'est le code le plus couramment employé pour définir les zones d'adresse dans les systèmes à base de microprocesseurs ou à base de microcontrôleurs). La plupart du temps, ce pointeur est simplement incrémenté chaque fois que l'on va chercher dans la mémoire programme un octet du programme (signalons qu'une instruction comporte de 1 à 3 octets). Toutefois, le contenu du pointeur est complètement modifié lorsque l'on doit effectuer un branchement (saut ou sous-programme) ou lorsqu'on effectue un retour au programme principal après exécution d'un sous programme. Pour permettre au microcontrôleur de reprendre l'exécution à l'endroit du saut, dans le cas d'appel à des sous-programmes, le processeur sauvegarde dans la pile l'adresse de retour avant d'effectuer le saut. Le processeur C51 utilise à cet effet le pointeur de pile SP (Stack Pointer) qui pointe automatiquement sur la dernière adresse sauvegardée. Il est incrémenté lors d'une sauvegarde d'adresse ou d'une donnée et décrémenté lors de la récupération d'une adresse ou d'une donnée. Par défaut le pointeur de pile est initialisé avec la valeur 07h à la mise sous tension (figure 4.2). Ce pointeur peut être par la suite modifié par le programme d'application. En ce qui concerne les données de type XDATA (données se trouvant dans la mémoire externe), le 8051 utilise le pointeur DPTR (Data Pointer) pour lire ou écrire une donnée en mémoire externe.

V.4.3 Traitement des instructions.

Le processeur C51 reprend partiellement les principes des processeurs RISC (décodeur d'instructions entièrement câblé et jeu d'instructions réduit). Il exécute 70% des instructions en un cycle machine, et les autres en deux cycles (hormis celles de multiplication et de division qui s'exécutent en quatre cycles). Le jeu d'instructions est un jeu réduit (101 instructions). Les instructions sont organisées autour d'un accumulateur et de registres (quatre banques de huit registres). La particularité du contrôleur est d'affecter une adresse mémoire à tous les registres (dont l'accumulateur), auxquels on peut donc avoir accès directement, ou par l'intermédiaire de leur adresse. Il n'y a pas d'instruction spécifique pour les registres de contrôle des périphériques intégrés. On y accède exclusivement par le mode d'adressage direct. La cadence de traitement des instructions est piloté par une horloge ; le 8051 contient tous les éléments de l'horloge, à l'exception des

composants qui déterminent la fréquence d'oscillation : généralement un quartz, mais un circuit RC peut aussi être utilisé si une connaissance précise de la fréquence n'est pas nécessaire.

V.5. Jeu d'instructions du processeur C51.

Il est le même pour tous les membres de la famille C51 et est optimisé pour les systèmes de contrôle 8-bits, d'où un très grand nombre de fonctions de manipulation de bits puisque de nombreux dispositifs de contrôle ont un fonctionnement à 2 états (ouvert-fermé ou tout ou rien). Nous donnons dans cette section, le jeu d'instructions du 8051 en employant la syntaxe du constructeur, Intel :

- Rr : registre de travail R0 à R7 du groupe sélectionné
- direct : adresse directe (RAM ou SFR)
- @Ri : case de RAM pointée par R0 ou R1
- @DPTR : case mémoire pointée par le DPTR
- #data : donnée immédiate 8 bits
- #data16 : donnée immédiate 16 bits
- bit : adresse bit (dans les 16 octets pour variables booléennes et les SFR)
- addr16 : adresse de destination pour les branchements
- rel : adresse relative pour les branchements ; l'adresse est indiquée en complément à 2, de façon à pouvoir effectuer des sauts en avant ou en arrière (les nombres de 128 à 255 sont considérés comme des nombres négatifs)

Remarque : Nous donnons en Annexe N°1 tous les détails concernant le jeu d'instructions du processeur C51

V.5.1.Instructions de transfert.

Les adresses de source et de destination peuvent être l'Acc, un registre de travail, une case de RAM interne ou externe, un port d'E/S ;

a) Transferts en RAM interne:

La structure des instructions est : **MOV destination, source** copie l'octet de l'adresse source à l'adresse de destination

```
MOV A,R7          ;copie le contenu de R7 dans l'Acc

MOV 70h,A         ;copie le contenu de l'Acc dans la case de RAM interne à l'adresse 70h

MOV A, @R1 ;copie le contenu de la case dont l'adresse est en R1 dans l'Acc

MOV 20h, 71h      ;copie le contenu de la case 71h de RAM interne dans la case 20h
```

b) Transferts de et vers la RAM externe:

La structure des instructions est : **MOVX destination, source**

MOVX A, @DPTR	;copie le contenu de la case dont l'adresse se trouve dans le DPTR dans l'Acc
MOVX @DPTR, A	;copie le contenu de l'Acc dans la case dont l'adresse se trouve dans le DPTR

V.5.2. Instructions arithmétiques.

D'une façon générale, elles utilisent l'Acc pour stocker une des deux données de l'opération, ainsi que le résultat de l'opération après exécution de l'instruction. Des indicateurs d'état sont positionnés automatiquement en fonction du résultat de l'opération :

- l'indicateur de dépassement (*Carry*) est mis à 1 si l'addition provoque un report au 9ème bit ;

- l'indicateur de semi-dépassement (*Half-Carry*) est mis à 1 si une addition provoque un report au 5e bit (cet indicateur est utilisé par l'instruction DAA, qui permet de faire l'addition de deux nombres en code DCB, décimal codé binaire) ;

- l'indicateur de parité est mis à 1 si la parité du nombre dans l'Acc est impaire ;

- l'indicateur d'*overflow* est positionné lors de certaines opérations arithmétiques.

Exemples :

ADD A, R5	;ajoute le contenu de R5 à l'Acc
ADD A,#17h	;ajoute 17h au contenu de l'Acc
SUBB A,R2	;soustrait à l'Acc le contenu de R2
MUL AB	;effectue la multiplication des nombres placés dans l'Acc et dans le registre B, les ;8 bits les moins significatifs sont placés dans l'Acc, les plus significatifs dans B
DIV AB	;divise le contenu de l'Acc par le contenu de B, place le quotient en A et le reste en B

INC R0	;incrémente le contenu de R0 sans modifier l'Acc

V.5.3. Instructions logiques.

Les opérations logiques sont effectuées entre bits de même poids, il n'y a pas d'interaction entre bits de poids différent et les indicateurs d'état ne sont pas modifiés. Les opérations ET, OU et XOU sont disponibles, ainsi que l'inversion ; pour les opérations NON-ET, NON-OU, NON-XOU, ils nécessitent deux étapes, d'abord l'opération directe, puis l'inversion de tous les bits.

Exemples :

CLR A	; met l'Acc à 0
CPL A	; complémente tous les bits de l'Acc
ANL A, 35h	; réalise une fonction ET entre les bits de l'Acc et ceux de la case 35h

V.5.4.Instructions booléennes.

Il s'agit ici d'une innovation par rapport aux microprocesseurs, qui traitent toujours les données par octets. Le 8051 contient un processeur complet agissant sur des données d'1 bit, aussi appelées variables booléennes. C'est le *Carry* qui joue le rôle d'Acc pour ces opérations.

Le RAM interne du 8051 contient 128 cases pouvant être adressées individuellement. Un certain nombre de bits des SFR sont aussi adressables individuellement, en particulier les bits correspondant aux ports de sortie.

Exemples :

SETB 38h	;positionne à 1 le bit 38h
CLR C	;met à 0 le Carry
ORL C,20h	;réalise un OU logique entre C et le bit 20h
MOV C,45h	;copie le contenu du bit 45h dans C.

V.5.5.Instructions de branchement.

Rappelons que les instructions de branchement permettent de rompre la séquence normale d'exécution d'un programme. On distingue :

- les sauts : on saute d'un endroit du programme à un autre, sans espoir de retour ;

- les sous-routines ou sous-programmes : on part exécuter un sous-programme, puis on revient à l'endroit d'où l'on était parti (l'adresse de retour est sauvegardée dans la pile).

Exemples :

LJMP addr16	;provoque un saut à l'adresse indiquée
LCALL addr16	;provoque l'exéciution du sous-programme qui commence à addr16
JZ F8h	;recule dans le programme de 8 pas si le contenu de l'Acc est nul (Jump *if Zero*)
CJNE A,#20h,F8h	;il s'agit ici d'une opération double : on compare les contenus de ;l'Acc et de la case 20h on recule de 8 pas dans le programme s'ils ne sont pas égaux ;(Compare and Jump if Not Equal).

Remarque : seuls les sauts peuvent être conditionnels avec le 8051.

V.6 les Ports d'E/S.

Les ports d'E/S permettent aux systèmes à microprocesseur ou microcontrôleur de communiquer avec le monde extérieur : recevoir des informations, qu'il va ensuite traiter et piloter les périphériques : témoins lumineux, moteurs, relais, convertisseurs N/A etc.

La famille 8051 a été conçue pour des applications nécessitant peu de mémoire ROM et RAM. Ainsi, le 8051 lui-même comporte une ROM de 4Ko et seulement 128 octets de RAM ; et encore faut-il tenir compte que la RAM est utilisée par les registres de travail, la pile et les 16 octets (de 20h à 2Fh) adressables par bit. Lorsque l'application peut se satisfaire de ces tailles mémoire, 32 lignes d'E/S sont disponibles, organisées en 4 ports de 8 lignes ; les adresses sont les suivantes : port 0 : 80h ; port 1 : 90h ; port 2 ; A0h ; port 3 : B0h. Les circuits internes associés aux différentes lignes sont légèrement différents. Prenons l'exemple du port1.

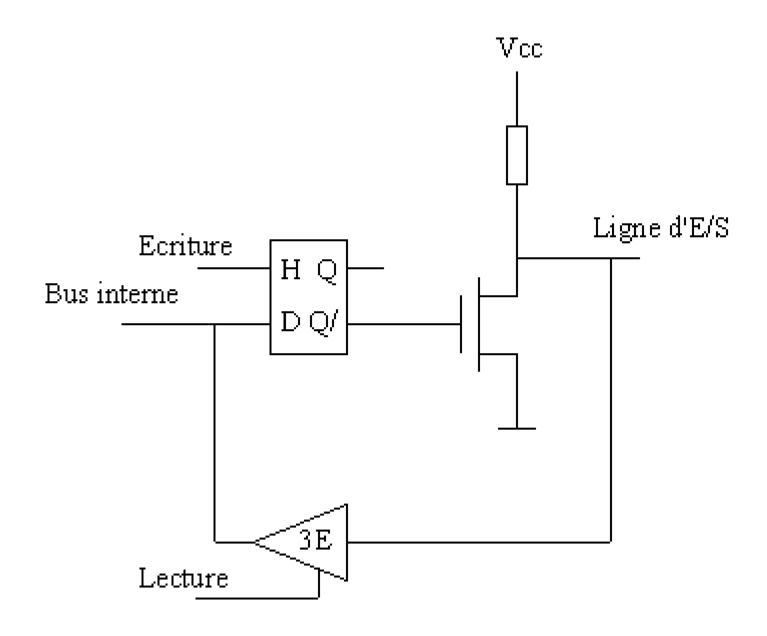


Figure 4.10 Structure d'une ligne du port 1 d'un MC 8051

Pour utiliser une ligne en ligne de sortie, on écrit le bit désiré dans la bascule D. La sortie Q/ (inverse de Q) pilote la grille d'un mosfet à enrichissement. Si Q est à 0, Q/ est à 1, le mosfet conduit et la ligne de sortie est tirée à 0. Si Q est à 1, Q/ est à 0, le mosfet est bloqué et la ligne de sortie est tirée vers Vcc par la résistance de rappel (*pull-up*). Pour utiliser la ligne en ligne d'entrée, il faut écrire un 1 dans la bascule D (c'est d'ailleurs l'état par défaut à la mise sous tension). Le circuit extérieur peut tirer la ligne à 0; l'état de la ligne apparaît sur le bus de données interne lorsqu'on fait une lecture du port. Lorsque l'on utilise des boîtiers de mémoire externe, le nombre de ports d'E/S disponibles est réduit à 2, en effet, les lignes de ports sont utilisées dans cette configuration comme bus d'adresses et de données multiplexés;

Les ports sont gérés comme des registres 8 bits, en écrivant ou lisant l'état de toutes les lignes d'un port simultanément par les instructions de transfert, ou comme ensemble de bascules gérant chacune 1 bit, grâce aux instructions booléennes. Signalons encore que certaines lignes du port 3 ont une double fonction puisqu'elles peuvent être utilisées comme lignes de demande d'interruption (§ V.6), comme entrées des compteurs (§ V.7), comme entrée et sortie du port série (§ V.8) et comme lignes Read (lecture) et Write (écriture) du bus de commande lorsqu'on fait appel à la RAM externe.

V.7 Les interruptions.

Comme son nom l'indique, une interruption est un événement qui interrompt l'exécution du programme et provoque un saut vers une routine dite d'interruption. La routine d'interruption, appelée Interrupt Handler en anglais, est seulement exécutée lors de l'arrivée d'un événement bien précis.

Avec le 8051, plusieurs sortes d'interruptions sont possibles :

- Overflow d'un des timers

- Réception ou transmission d'un caractère sur le port série

- Activation d'événements extérieurs via les pins T0 ou T1

Le 8051 peut être configuré afin que lorsque l'un de ces événements arrive, le programme principal soit temporairement interrompu et le contrôle soit passé à une section spéciale du code qui effectue une fonction dédicacée à l'événement. Une fois cette routine effectuée, le contrôle est rendu au programme principal.

Il existe plusieurs types d'interruption. Chaque type est dédié à un événement bien précis est caractérisé par une adresse propre dans la zone mémoire programmes. Nous donnons dans le tableau suivant les différentes adresses des 5 sources d'interruptions que comprend le 80C51.

Interruption	Flag	Adresse
Externe 0	IE0	0003h
Timer 0	TF0	000Bh
Externe 1	IE1	0013h
Timer 1	TF1	001Bh
Port Série	RI/TI	0023h

Tableau 4.2. Les interruptions du 80C51

En consultant le tableau ci-dessus, on peut voir que quand le Timer 0 déborde ("overflow"), le microcontrôleur met le bit TF0 à 1 et le programme principal est suspendu. L'instruction suivante est directement exécutée à l'adresse 000Bh. A cette adresse doit donc figurer le code traitant l'interruption en question.

Par défaut, à la mise sous tension ou à la suite d'un RESET du microcontrôleur, toutes les interruptions sont inhibées. Cela signifie que, même si, par exemple, Le flag TF0 passe à 1, le 8051 n'exécutera pas la routine d'interruption. Le programme doit donc spécifiquement

signaler au 8051 quelles interruptions doivent être activées. Le programme d'application doit activer ou désactiver les interruptions en modifiant le registre IE (A8h). Ce registre IE est composé des bits suivants :

Bit	Nom	Adresse du bit	Fonction
7	EA	AFh	Activation globale des interruptions
6	-	AEh	Indéfini
5	-	ADh	Indéfini
4	ES	ACh	Activation interruption port série
3	ET1	ABh	Activation interruption Timer 1
2	EX1	AAh	Activation interruption externe 1
1	ET0	A9h	Activation interruption Timer 0
0	EX0	A8h	Activation interruption externe 0

Tableau 4.3. Registre de contrôle des interruptions du 80C51

Comme nous pouvons le constater, chaque interruption a son propre bit d'activation dans le registre IE. Pour activer une interruption, il suffit de mettre à 1 le bit correspondant à cette interruption dans le registre IE. Par exemple pour le Timer 1 :

MOV IE,#08h

ou

SETB ET1

Les deux instructions ci-dessus mettent à 1 le bit 3 de IE, activant donc l'interruption du Timer 1. Bien sûr, pour que les instructions soit vraiment activées, il faut mettre à 1 le bit 7 et autoriser de manière globale les interruptions. EA est vraiment utile lorsque des passages du code sont critiques en ce qui concerne leur durée d'exécution. On inhibe alors les interruptions pour un temps. En ce qui concerne la détection des interruptions, le 8051 évalue automatiquement après chaque instruction si une interruption doit avoir lieu ou non. Il effectue le test dans l'ordre suivant :

- Interruption Externe 0
- Interruption Timer 0
- Interruption Externe 1
- Interruption Timer 1
- Interruption Port Série

Cela signifie que si une Interruption Port Série intervient en même temps qu'une Interruption Externe 0, celle-ci sera exécutée en premier lieu et quand elle sera terminée, le 8051 s'occupera de l'Interruption Port Série. Ceci nous ramène à parler des priorités dans la gestion des interruptions. Le 8051 offre deux niveaux de priorités : niveau haut et niveau bas. Rien de tel qu'un bon exemple pour comprendre l'utilité des priorités. Supposons que nous avions activé l'interruption du Timer 1 qui sera appelé automatiquement chaque fois que le Timer 1 déborde. De la même manière nous avions activé l'interruption du port série qui sera appelé chaque fois qu'un caractère sera reçu via le port série. Si nous considérons que la communication avec le port série est plus importante que l'interruption du Timer 1, il faut accorder une priorité plus élevée au port série. Cela veut dire si une interruption Timer 1 est en cours, l'interruption port série peut prendre la main si un caractère arrive. Une fois celle-ci terminée, elle rend le contrôle au Timer 1 qui à son tour rendra la main au programme principal lors de son achèvement. Les priorités sont gérées par le registre IP (B8h). Le tableau suivant donne la composition du registre IP.

Bit	Nom	Adresse du bit	Explication de la fonction
7	-	-	Indéfini
6	-	-	Indéfini
5	-	-	Indéfini
4	PS	BCh	Priorité Port Série
3	PT1	BBh	Priorité Timer 1
2	PX1	BAh	Priorité interruption Externe 1
1	PT0	B9h	Priorité Timer 2
0	PX2	B8h	Priorité interruption Externe 0

Tableau 4.4. Le registre de gestion des interruptions

Nous allons décrire par la suite le principe de fonctionnement des interruptions. Quand une interruption est déclenchée, le 8051 effectue automatiquement les actions suivantes :

- Le PC (program counter) est sauvegardé dans la pile (stack)
- Les interruptions de même priorité ou de priorité inférieure sont bloquées
- Dans le cas d'interruptions Externes ou Timer le Flag d'interruption correspondant est mis à 1.

- L'exécution du programme est transférée vers l'adresse interruption dédicacée.

Le programme d'interruption finit lorsqu'il rencontre l'instruction RETI (Return From Interrupt). Le 8051 restaure alors l'adresse de retour de la pile et reprend l'exécution à l'endroit où il a été arrêté par l'événement.

V.8 Les compteurs/temporisateurs.

Le 8051 comporte deux compteurs/temporisateurs. Ces périphériques sont appelés ainsi car ils peuvent être utilisés :

- pour compter les impulsions appliquées à une broche du microcontrôleur ; on parle alors de **compteur** (*counter*) ;

- pour compter des impulsions provenant de l'horloge du microcontrôleur; on parle alors de **temporisateur** (*timer*) puisque dans ce mode le microcontrôleur peut mesurer des intervalles de temps ou générer des délais précis.

Chaque C/T comporte deux compteurs binaires de 8 bits. Différents modes de fonctionnement sont possibles : les plus utilisés sont :

- le mode 1 : les compteurs sont mis en cascade pour former un compteur 16 bits ; une demande d'interruption a lieu lorsqu'il y a dépassement de la capacité du compteur, c'est à dire lorsque le compteur passe de FFFFh à 0000h ;

- le mode 2 : l'un des compteurs est utilisé comme compteur, l'autre comme registre dans lequel on peut écrire un nombre de 8 bits quelconque ; lorsqu'il y a dépassement de capacité du compteur, un rechargement automatique de celui-ci est effectué par le registre; cette technique permet donc d'obtenir des demandes d'interruption régulièrement espacées si le compteur reçoit des impulsions d'horloge.

On peut choisir d'appliquer à chaque compteur :

- soit des impulsions provenant d'une broche externe T0 ou T1 ;

- soit des impulsions provenant du circuit d'horloge, après passage par un diviseur de fréquence par 12.

L'activation ainsi que la désactivation de ces deux Timers sont contrôlés par deux bits du registre **TCON** (Timer CONtrol) adressable au niveau du bit **:** TR0 pour Timer0 et TR1 pour Timer1 **(*figure .4.11*).**

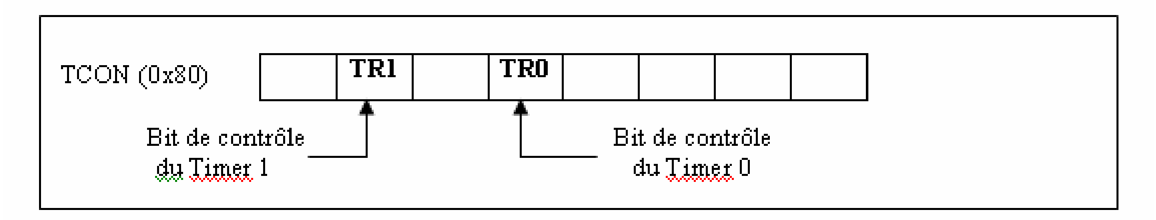


Figure 4.11. Registre de contrôle TCON

La mise à 1 de ces bits permet d'activer le Timer correspondant qui commence alors à s'incrémenter; la mise à 0 de ces bits désactive le Timer correspondant qui s'arrête et garde le même contenu. Ces deux bits peuvent être accédés par les instructions : TR0 = 1 ; TR1 = 0 (compilateur 'C') ;

Les Timers 0 et 1 peuvent être configurés indépendamment l'un de l'autre (exception mode 4) à partir d'un même registre de configuration TMOD (Timer Mode) : les 4 bits poids fort pour la configuration du Timer1, les 4 bits poids faible pour le Timer0 (figure 4.12)

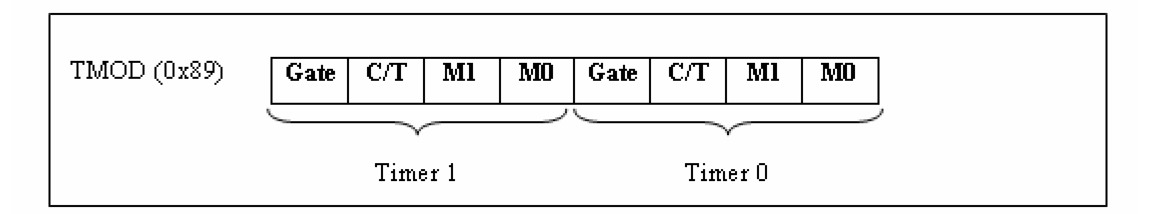


Figure 4.12 Le registre TMOD

Les paragraphes suivants décrivent les différentes configurations possibles du Timer0. Mais ce qui est valable pour le Timer0 l'est également pour le Timer1 (qui est généralement configuré pour générer la vitesse de transmission du port série (voir § V.8).

a) Mode Temporisation

Pour configurer le Timer0 dans ce mode, les bits Gate et C/T du registre TMOD doivent être à 0. M1 et M0 servent alors à choisir la valeur maximale que le Timer peut atteindre avant de retourner à 0: on parle de débordement. Dans ce mode, le Timer0 est incrémenté à une fréquence égale à 1/12 de la fréquence horloge du microcontrôleur (figure 4.13)

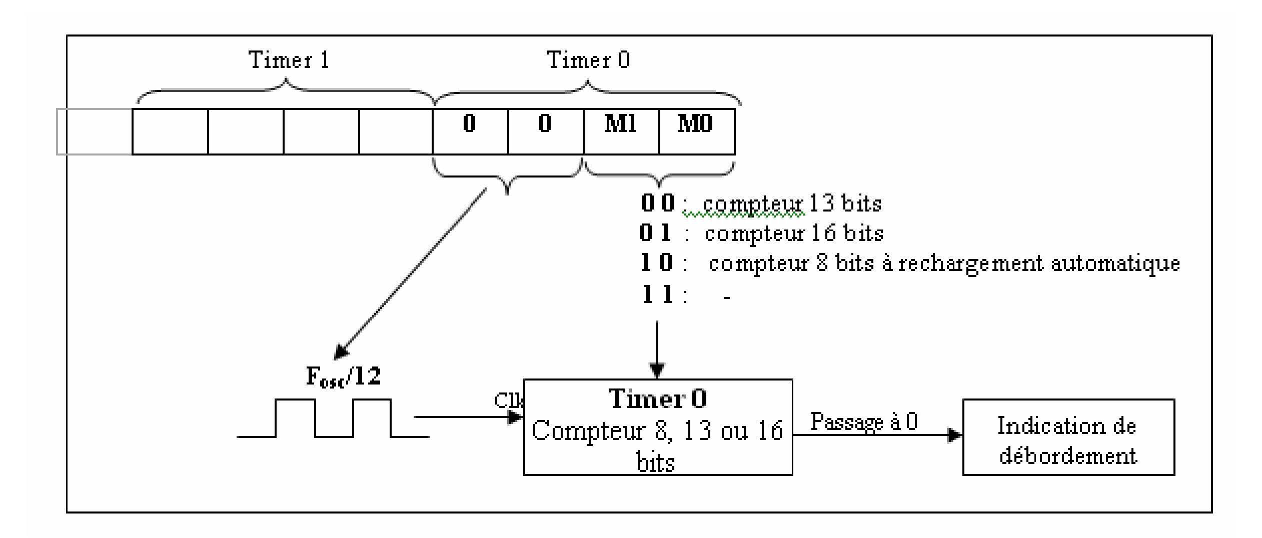


Figure 4.13. Timer 0 en mode Temporisateur

Les valeurs maximales atteintes par le Timer sont donc respectivement ($2^{13}-1$ = 8191 pour le mode 13 bits, $2^{16}-1$ = 65535 pour le mode 16 bits et $2^{8}-1$ = 255 pour le mode 8 bits). L'activation du Timer associée à l'autorisation de l'interruption relative à l'indication de débordement, offre un moyen très précis pour les applications nécessitant la gestion d'une base de temps (exemple : horloge temps réel, acquisition périodique).

Exemple : on cherche a changer l'état logique du bit poids fort du port parallèle P4 chaque 1/100 sec : à cette fin on utilisera le Timer0 de telle sorte à avoir une interruption toutes les 1/100 sec. C'est la routine associée à cette interruption qui servira à changer l'état de ce bit. (on suppose que la fréquence d'horloge du microcontrôleur est de 12 Mhz). On doit tout d'abord déterminer le nombre d'incrémentations nécessaires pour avoir la fréquence d'interruption voulue et par conséquent le mode de fonctionnement du Timer (13, 16 ou 8 bits) ainsi que la valeur d'initialisation des registres TH0 et TL0.

On a d'une part : la fréquence d'incrémentation = 12/12Mhz = 1Mhz ; ce qui correspond à une période d'incrémentations = 1µsec.

D'autre part on a : période des interruptions : 1/100 sec = 10000 µsec. Donc le nombre d'incrémentations est 10000 µsec/1µsec = 10000 (> 8192 et > 256).

➡ **Timer0 doit être donc configuré comme compteur 16 bits (M1M0 = 01)**

La valeur d'initialisation est alors égale à : 2^{16}-10000 = 55536 = 0xDF80.

➡ **TH0 = 0xDF et TL0 = 0x80**

#include <reg51.h>	
void ***centieme_seconde*** (void) **interrupt 1**	// C’est la routine associée au débordement du Timer0 qui a le //numéro 1
{	
P4 = P4 ^ 0x80 ;	// Inverser l’état logique du bit poids fort de P4.
TH0 = 0xDF ;	// recharger le Timer0 (TH et TL0). Le rechargement n’est pas obligatoire dans le mode // 8 bits ; il est automatique (géré matériellement).
TL0 = 0x80 ;	
}	
void **main** (void)	
{	- | - | - | - | 0 | 0 | 0 | 1
TMOD = 0x01 ;	// TMOD (Gate et C/T = 0 ; M1 M0 = 01)
TH0 = 0xDF ;	// Chargement des registres poids fort TH0 et poids faible TL0 du Timer0.
TL0 = 0x80 ;	
EA = 1 ;	// Autorisation générale des interruptions.
ET0 = 1 ;	// Autorisation de l’interruption associée au débordement du Timer0.
TR0 = 1 ;	// Activer le Timer0 (incrémentation avec une cadence $f_{osc}/12$).
While (1)	
{	
}	
}	

b) Mode Comptage

La seule différence entre ce mode et le mode temporisation est la source de la cadence d’incrémentation : Cette source dans ce mode est issue des fronts descendants au niveau de la broche T0 du microcontrôleur pour le Timer0 et T1 pour le Timer1. Pour configurer le Timer0 en mode comptage C/T doit être à 1 ; les autres bits gardent le même rôle décrit en mode temporisation (figure 4.14).

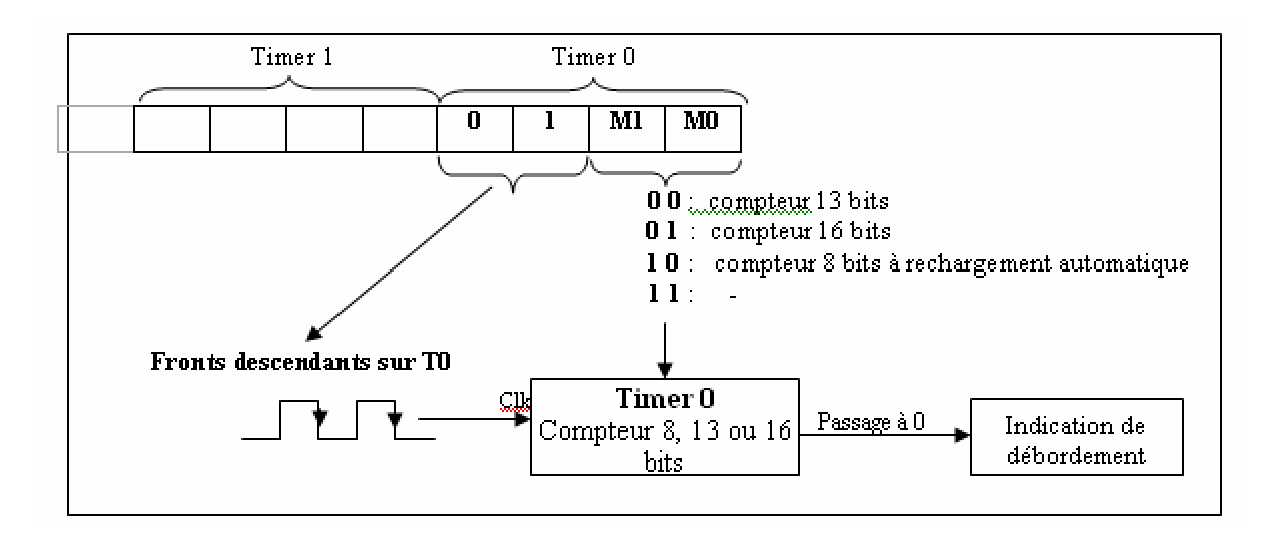


Figure 1.14. Timer0 en mode comptage

<u>Exemple</u> : Diviseur de fréquence

On applique un signal carré S_f de fréquence **f** sur la broche T0 du microcontrôleur et on génère un signal carré de fréquence f/10 sur le bit poids faible du port parallèle P4. Ceci est réalisé en générant une interruption toutes les **5** (10/2) **périodes** du signal S_f **:** Le timer0 sera configuré en mode 8 bits (5<256) donc M1 M0 = 10 et initialisé avec la valeur 256-5=251= 0xFB. La routine associée à l'interruption assure le changement d'état de la sortie (P4.0) (figure 4.15).

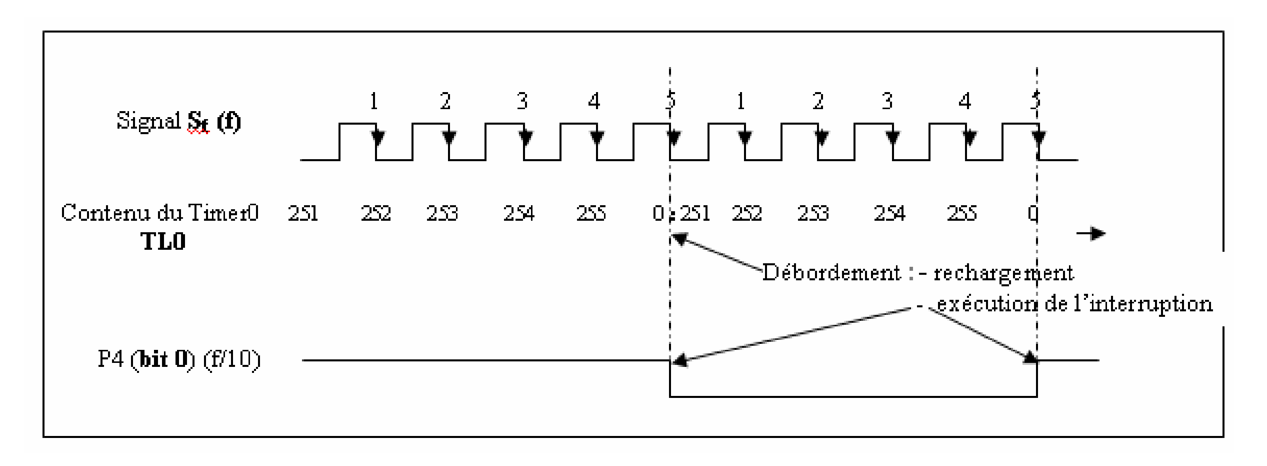


Figure 4.15: Diviseur de fréquence (utilisation du Timer0)

#include <reg51.h>	
void ***centieme_seconde*** (void) **interrupt 1**	// C'est la routine associée au débordement du Timer0 qui a le //numéro 1 (voir document).
{	
P4 = P4 ^ 0x01 ;	// Inverser l'état logique du bit poids faible de P4.
}	
void **main** (void)	- - - - 0 1 1 0
{	
TMOD = 0x06 ;	// TMOD (Gate=0 et C/T = 1 ; M1 M0 = 10)
TL0 = 0xFB ;	// Chargement des registres poids fort TH0 et poids faible TL0 du Timer0.
EA = 1 ;	// Autorisation générale des interruptions.
ET0 = 1 ;	// Autorisation de l'interruption associée au débordement du Timer0.
TR0 = 1 ;	// Activer le Timer0 (incrémentation avec la cadence du signal S_f).
While (1) { }	
}	

V.9 Le port série

Pour certaines applications (transfert de données à distance, commande de modems,...), il est utile de disposer en plus des ports d'E/S parallèles, d'un port d'entrée/sortie série ou sériel (*Serial Input/Output Port*). Le principe des liaisons série est très simple : plutôt que d'envoyer simultanément 8 bits en parallèle sur 8 lignes, on envoie les 8 bits l'un après l'autre sur une seule ligne. Le 8051 dispose d'un port série bidirectionnel utilisant une ligne pour l'envoi des données, TxD, et une ligne pour la réception des données, RxD.

Plusieurs modes de fonctionnement sont possibles. Le mode le plus utilisé emploie le temporisateur 1 en mode de rechargement automatique pour définir la cadence d'envoi des bits (*baud rate*). Les cadences standard vont de 300bps (bits par seconde) à 19200bps.

Pour envoyer un caractère, il suffit de l'écrire dans le registre Sbuf d'émission ; cela enclenche automatiquement la procédure d'envoi du caractère :

- le premier bit s'appelle le bit Start, toujours à 0 ;

- il est suivi par les 8 bits de l'octet à transmettre ;

- on termine par le bit Stop, toujours à 1 ; la présence des bits Start et Stop assure qu'il y a toujours un front descendant en début de transmission d'un octet ; la ligne reste à 1 jusqu'à l'octet suivant ;

Sur la figure suivante, nous présentons le chronogramme de fonctionnement de la liaison série. Dans cet exemple nous avons choisi d'envoyer le caractère E dont le code ASCII est 69 (01000101) avec un bit de stop et sans bit de parité

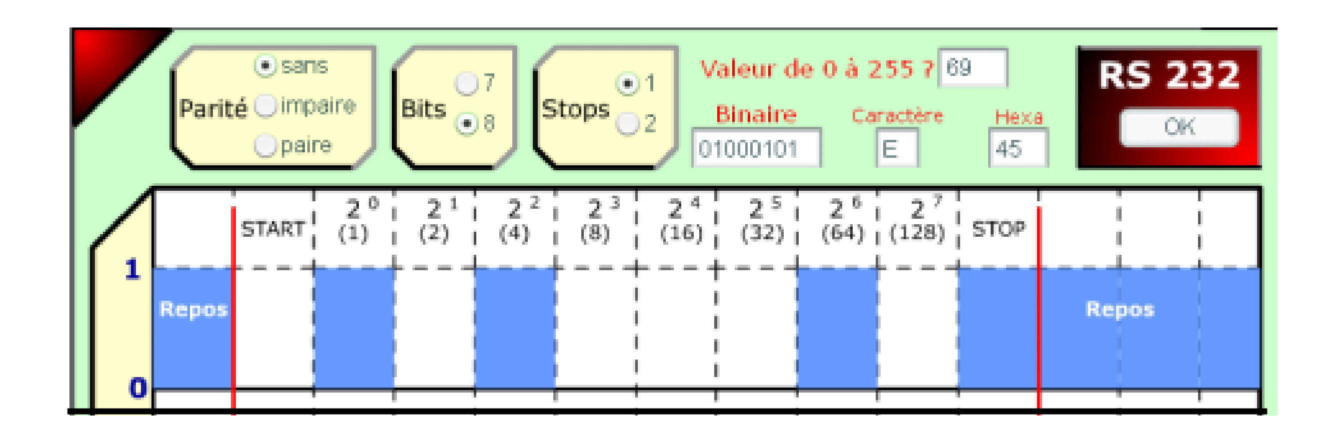


Figure 4.16 Chronogramme de fonctionnement de la liaison série

Lorsque le registre Sbuf est vide, le microcontrôleur déclenche une demande d'interruption ; le processeur enverra alors l'octet suivant, si nécessaire ; Il est possible aussi d'envoyer des mots de 9 bits plutôt que 8 ; ceci permet d'accoler aux 8 bits un bit de parité pour le contrôle des erreurs, ou de créer un petit réseau avec plusieurs processeurs. Lors de la réception d'un caractère, la procédure est enclenchée par la détection d'un flanc descendant à la broche RxD. Les 8 ou 9 bits sont placés dans le registre Sbuf de réception (curieusement, on utilise le même nom et la même adresse, 99h, pour les deux registres Sbuf ; quand on écrit à cette adresse, l'octet est stocké dans le registre d'émission, quand on lit, c'est le contenu du registre de réception). Lorsque tous les bits sont présents, on déclenche une demande d'interruption. Il faut que le microcontrôleur vienne lire l'octet reçu avant la fin de la

réception de l'octet suivant, sinon le premier est perdu. L'émetteur et le récepteur peuvent travailler simultanément. Les demandes d'interruption sont combinées. Donc, lorsque l'on envoie et reçoit simultanément des octets, le processeur doit déterminer lors de chaque demande d'interruption si elle émane de l'émetteur ou du récepteur. Cela se fait en allant lire les bits TI et RI du registre Scon (*Serial Port Control Register*).

V.9.1 Configuration de l'interface série .

L'interface série peut fonctionner selon 4 modes ; la sélection de l'un de ces modes se fait par les deux bits poids fort d'un registre à fonction spéciale (0x98) adressable au niveau du bit: **S0CON** (**S**erial **CON**figuration **0**). Ce registre contient également les bits d'indication d'émission, d'indication de réception et de validation de réception :

SM0	SM1	-	REN	-	-	TI	RI

Figure 4.17 Registre S0CON

SM0 et **SM1** spécifient le mode de fonctionnement du port série :

SM0	SM1	Mode	Description	Vitesse
0	0	0	Registre à décalage	$F_{osc}/12$*
0	1	1	UART 8 bits	Variable
1	0	2	UART 9 bits	$F_{osc}/64$ ou $F_{osc}/32$*
1	1	3	UART 9 bits	Variable

REN = 1 : réception sur le port série est activée.
= 0 : réception sur le port série inhibée.

TI : c'est l'indicateur de transmission, il est mis automatiquement à 1 quand la donnée a été complètement envoyée . Il doit être remis à 0 par programme.

RI : c'est l'indicateur de réception, il est automatiquement mis à 1 à la réception d'une donnée complète. Il doit être remis à 0 par programme.

* F_{osc} est la fréquence horloge du microcontrôleur.

V.9.2 Vitesse de transmission

Deux parmi les 4 modes de fonctionnement du port série sont à vitesse variable : dans ces deux modes la vitesse est déterminée par la fréquence de débordements (dépassement de 255 et passage à 0) du Timer1, qui est constitué de deux registres 8 bits TH1 et TL1, et qui doit être configuré en mode temporisateur 8 bits à rechargement automatique à travers le registre **TMOD** (0x89).

On présente les bits intervenant dans la configuration du timer1 :

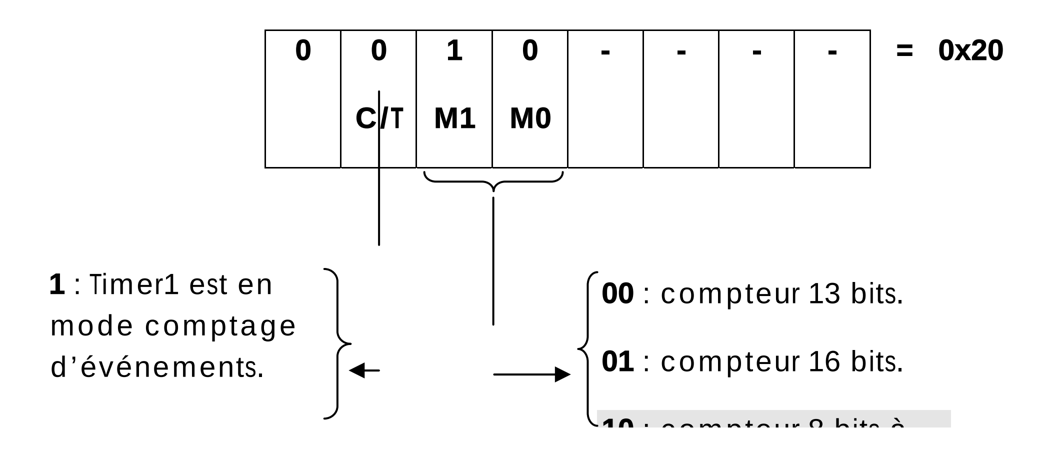


Figure 4.18 Registre TMOD

Ce timer peut être activé ou arrêté suivant l'état d'un bit du registre **TCON** adressable au niveau bit (Timer **CON**trol) : il s' = 1 : Timer1 actif.

= 0 : Timer1 arrêté.

Dans le cas de la configuration donnée sur la figure 4.17, la valeur de TH1 est chargée dans TL1 qui est incrémenté avec une fréquence F_{osc} /12*32. Une fois TL1 repasse à 0, il y'a indication de débordement ainsi qu'un rechargement de TL1 avec la valeur de TH1, et le cycle recommence : c'est la fréquence de débordement du timer qui fixe la vitesse de transmission **(figure 4.19).**

Exemple :

On veut établir une liaison série avec une vitesse de 9600 bits /seconde avec un microcontrôleur ayant une fréquence d'horloge de 12 Mhz.

- Déterminer la valeur que doit contenir TH1.

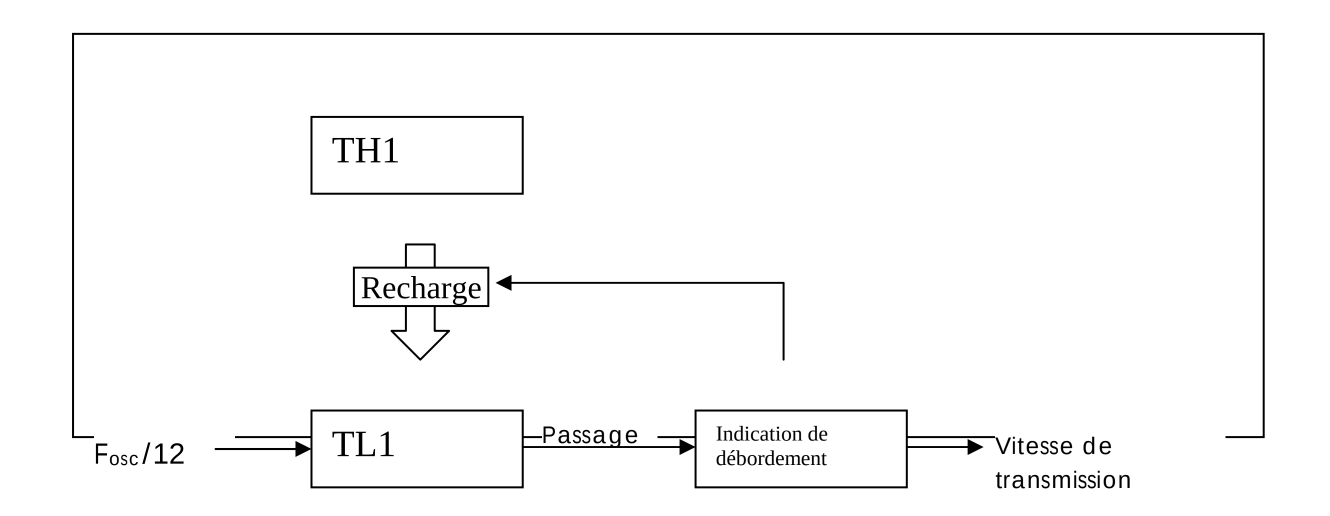


Figure 4.19. Génération de la vitesse de transmission du port série

Solution :

- La fréquence d'incrémentation de TL 1 est 12 Mhz/12*32 =31250 Hz .

⟹ TL1 est incrémenté de 1 toutes les 1/31250 Hz = 32 µsec.

- Pour avoir une vitesse de 9600 bits/sec, il faut avoir 9600 débordements par seconde.

  ⟹ TL1 doit déborder toutes les 1/9600 secondes ~ 104 µsecondes.

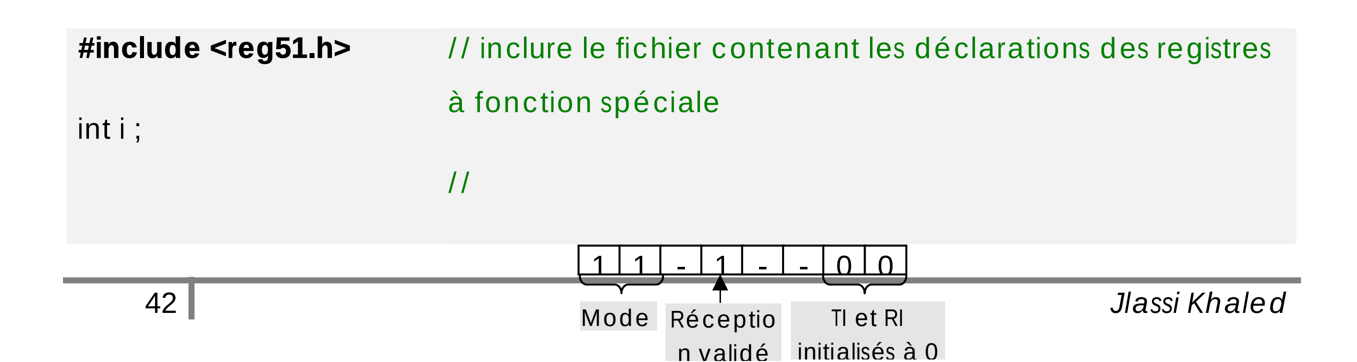
Donc TL1 doit être incrémenté 104 fois* (104 µsec / 32 µsec) = 3,25 ~ 3 avant de déborder et par conséquent contenir la valeur 256- 3 = 253 (0xFD). C'est cette valeur qui doit être stockée dans TH1.

Remarque : Ce résultat est obtenu également en divisant la fréquence d'incrémentation par la vitesse de transmission désirée (31250/9600 = 104).

Application :

Le code suivant écrit en langage 'C' permet :

- la configuration de l'interface série en mode UART 9 bits (mode 3) avec une vitesse de transmission de 9600 bits/sec avec validation de la réception. (F_{osc} = 12 Mhz)
- puis la réception de 10 données.
- suivie par la transmission de 10 données.
- et enfin la désactivation de la réception.



```
#include <reg51.h>      // inclure le fichier contenant les déclarations des registres
                        à fonction spéciale
int i ;
                        //
```

```
char datas[10] ;

void Main (void) {

S0CON = 0xD1 ;
                        // S0CON =
                        //

TMOD = 0x20 ;
                        // Timer1 configuré en temporisateur 8 bits.
                        // TMOD =

TH1 =  0x98 ;
                        // Voir exemple page 3.

TR1 = 1 ;
                        // Lancer le Timer1.

for (i=1 ; <=10 ; i++ )
                        // Réception de 10 données.
{
while ( !RI) ;
                        // Attendre jusqu'à ce que l'indicateur de réception passe
datas [ i ] = S0BUF ;   à 1.
RI = 0 ;                // Lire le donnée à partir de S0BUF.
}                       // Remettre RI à 0.
for (i=1 ; <=10 ; i++ )
{                       // Transmission de 10 données.
while ( !TI) ;
TI = 0 ;                // Attendre l'émission complète de la donnée précédente
```

```
S0BUF = datas [ i ] ;          (passage de TI à 1).

}                              // Remettre TI à 0.

REN = 0 ;                      // Mettre  la donnée à envoyer dans le registre de
                               transmission ( S0BUF ).
}

                               // désactiver la réception.
```

Remarque : - Les bits non utilisés sont supposés mis à 0. (S0CON et TMOD).

- En utilisant les fonctions printf, scanf, etc..., on n’est plus obligé de gérer les bits TI et RI.

I. Configuration matérielle de la carte.

La carte qui sera utilisée pour illustrer nos exemples est à base d'un microcontrôleur fabriqué par Philips (80C552) et conçu autour du noyau 8051 (famille Intel). La carte d'étude est équipée de trois circuits mémoires : une EPROM (64 Koctets), une RAM sauvegardée par batterie (32 Koctets) et d'une RAM (32 Koctets). Ces circuits peuvent être configurés de plusieurs manières à travers 11 cavaliers (J1..J11), pour occuper des types (code, données) et des zones d'adressage différents.

La configuration utilisée est la suivante :

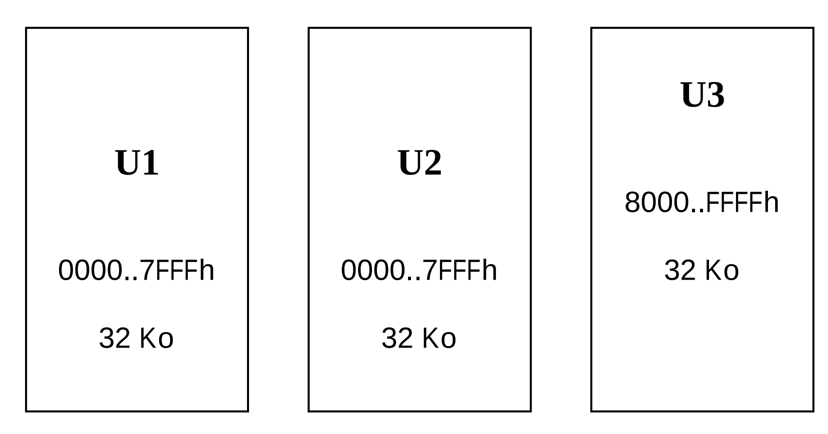


Figure 5.1. Configuration de la carte

- EPROM (U1) : seulement les 32 Koctets inférieurs sont utilisés et occupent la zone de code 0000..7FFFH (les 32 Koctets inf.). Cette zone contient le **moniteur** : un programme qui est lancée au démarrage de la carte et qui permet la visualisation des différents registres du microcontrôleur, du contenu des mémoires, la gestion de la liaison série ainsi que le chargement des programmes dans la zone code et leur exécution , c'est un mini-système d'exploitation de la carte. Bien sûr, cette zone ne peut pas être utilisée pour charger des programmes d'application.

- RAM (U2) : elle occupe les 32 Ko inférieurs de la zone DATA ou données (0000..7FFFH). La zone située de 7F00..7FFFH est utilisée par le moniteur. Le reste peut être utilisé par d'autres programmes.

- RAM sauvegardée par batterie (U3) : elle occupe les 32 Ko supérieurs des zones code et DATA en même temps. Elle peut être utilisée entièrement par les programmes, mais il faut faire attention pour que les données et le code ne soient pas placés dans des zones qui se chevauchent.

II. Le format Hexadécimal.

Un programme écrit, que ce soit en un langage évolué tel que C, ou en langage assembleur va être traité (compilation + édition des liens) pour produire enfin une suite de données binaires compatibles avec le jeu d'instruction du microcontrôleur.

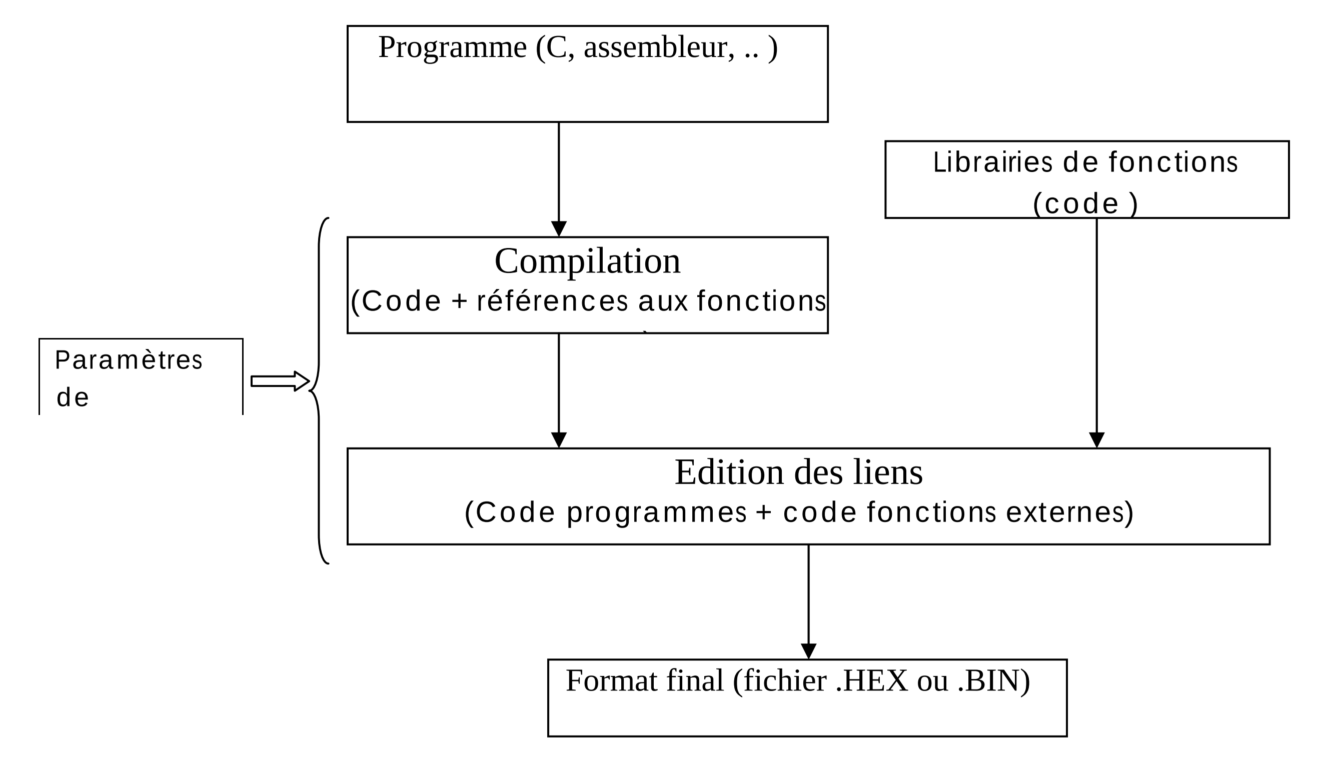
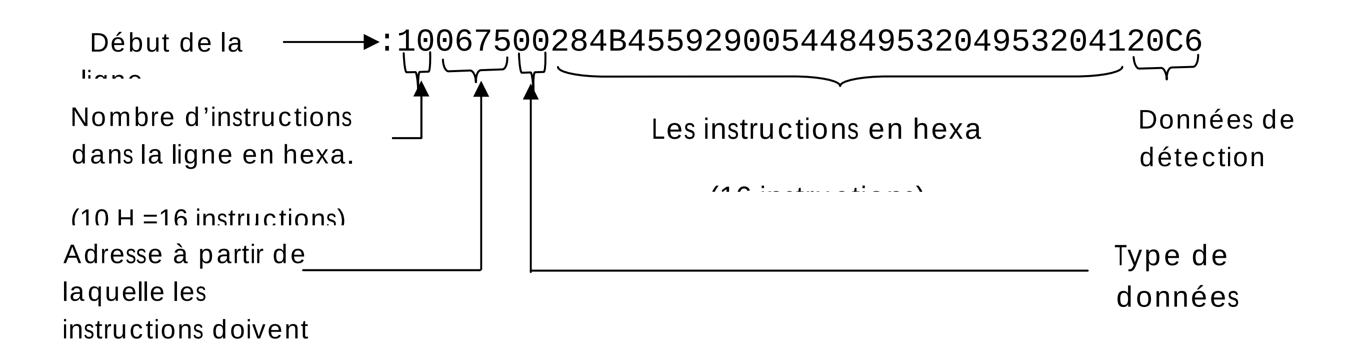


Figure 5.2. Génération des formats .BIN et .HEX

Le fichier généré peut être :

- soit de format .BIN, qui contient uniquement du code (image de la mémoire cible) et peut être implanté directement sur une EPROM qui équipera ensuite la carte microcontrôleur.
- soit de format .HEX qui contient en plus du code, les adresses à partir desquelles le code doit être logé, ainsi que des données de détection d'erreurs. Ce format est destiné à être transféré par liaison série ou autre à une carte μc équipée d'un moniteur pouvant gérer ce format (ce qui est notre cas).

Un fichier .HEX est composé d'une suite de lignes, en voici un exemple d'une ligne :



La dernière ligne d'un fichier .HEX étant (:00000001FF) et elle indique au moniteur la fin du fichier.

III. Connexion entre le PC et carte.

Pour pouvoir charger le programme dans la zone code de la carte après avoir généré le fichier .HEX, il faut disposer du coté PC :

- d'un port série libre.
- d'un logiciel permettant la transmission et la réception à travers le port série (on utilisera ***HYPERTERMINAL*** qui est inclus dans le CD d'installation de Windows) et étant configuré de la même manière que le moniteur de la carte : (9600,8,N,1)
 (vitesse = 9600bps; Nbre de bits données = 8 ; Parité :aucune ; Nbre bits de stop =1.

Enfin on a besoin d'un câble série pour connecter le port série du PC à celui du microcontrôleur. (voir **figure 5.3**)

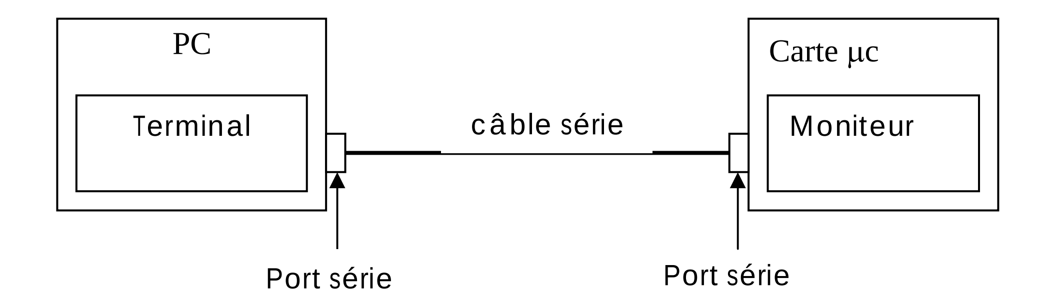


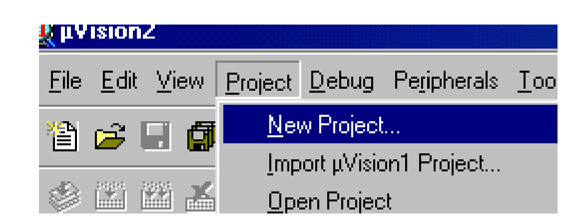
Figure 5.3. liaison série entre PC et carte

IV. L'environnement de développement.

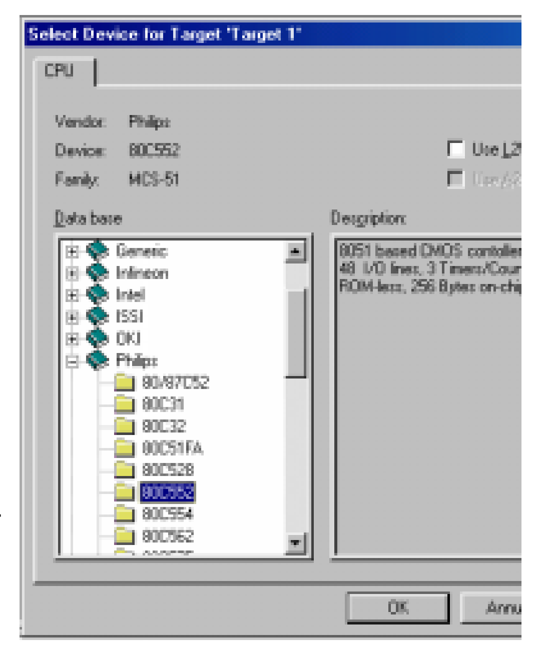
La saisie, la simulation et la compilation des programmes au cours de nos exemples seront réalisées dans l'environnement de développement Keil qui supporte en plus du langage assembleur, le langage C (que nous utiliserons) et comporte également un grand nombre de libraires (communication, calcul avec virgule flottante, etc..) .

Avant d'écrire des programmes, il faut commencer par créer un projet avec le menu

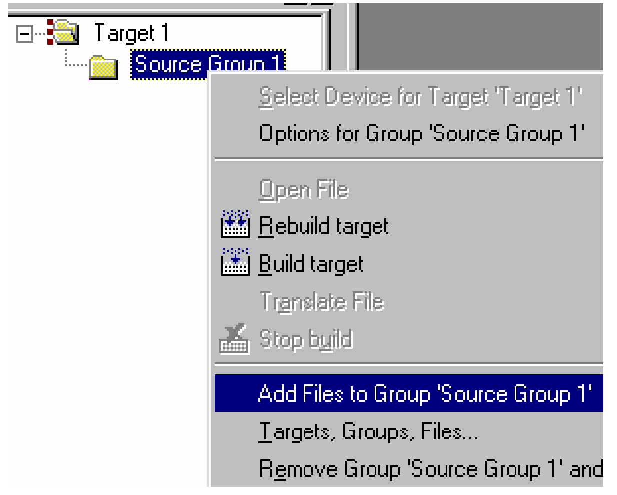
Project>>New Project



- Une fois on a choisi le chemin d'enregistrement et le nom du projet, il faut sélectionner le type de microcontrôleur utilisé parmi la liste supportée par Keil dans la fenêtre " ***Select Device for Target 'Target1'*** ".



- Sélectionner le constructeur (Philips) puis la référence du microcontrôleur (80C552) et valider.



Maintenant il faut créer un nouveau fichier avec le menu ***File>>New***, l'enregistrer en précisant l'extension (**.c** dans notre cas) et enfin la déclarer comme faisant partie du projet : sélectionner *'Source Group 1'* dans la fenêtre gauche , appuyer sur le bouton droite de la souris et cliquer sur '***Add files to….***'. Enfin sélectionner le fichier et l'ajouter (**Add**). (voir figure ci-contre)

Exemple :

Le programme qu'on va traiter au cours de notre exemple est assez simple, mais montre la richesse des librairies et la puissance de l'environnement Keil surtout dans les phases de débogage et simulation.

Le programme utilise deux fonctions de la librairie fournie :

- ***printf*** : qui permet de transmette des données vers la liaison série du microcontrôleur.
- ***Scanf*** : qui permet au microcontrôleur de recevoir les données à partir de son port série.

On commence par envoyer un message 'V*euillez choisir un chiffre*' et attendre jusqu'à recevoir une donnée. Si la donnée reçue est un chiffre, on envoie le chiffre en toutes lettres (exemple zéro pour 0, etc..). Sinon on envoie le message '*Erreur' :*

```
#include <reg552.h>                /* Fichier contenant les déclaration des différents
                                   registres du noyau 8051
                                   */

void Main(void) {

int i;                             /*Déclaration de variables        */

char datas[10]="bonjour!!!";       /* une variable de 10 caractères */

                                   /*

S0CON = 0x52 ;                       Configuration et Activation du Timer et  du
                                   registre SCONP(Serial Configuration)  pour les
                                   paramètres de  communication série :  9600
                                   ,8,N,1.
TMOD = 0x20 ;

                                   */
TH1 =  0xFD ;

                                   /*lancer le timer1*/
TR1 = 1 ;

                                   /* Envoi des caractères de la variable datas sous
for (i=0 ; i<10; i++ )             forme d'une suite de codes ASCII    */

{

while ( !TI) ;

TI = 0 ;

S0BUF = datas [ i ] ;

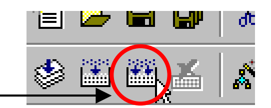
}

}
```

IV.1 Phase de simulation.

Une fois le programme saisi, nous pouvons lancer la compilation et l'édition des liens avec le menu suivant :

Project >> Build target ou ***Rebuild all target files*** si vous avez modifié plusieurs fichiers du projet. Ou utiliser l'icône correspondante



En bas de l'écran, apparaîtra un message indiquant que le programme a été compilé et affichant les erreurs s'il y en a. Dans notre exemple et avec un fichier tp1.c on devra retrouver le message suivant:

Build target 'target 1'

Compililng tp1.c….

Linking….

0 Error(s) , 0 Warning(s)

Maintenant, nous pouvons passer en mode débogage avec le menu ***Debug>>Start/Stop Debug Session***.

ou bien l'icône correspondante de la barre d'outils

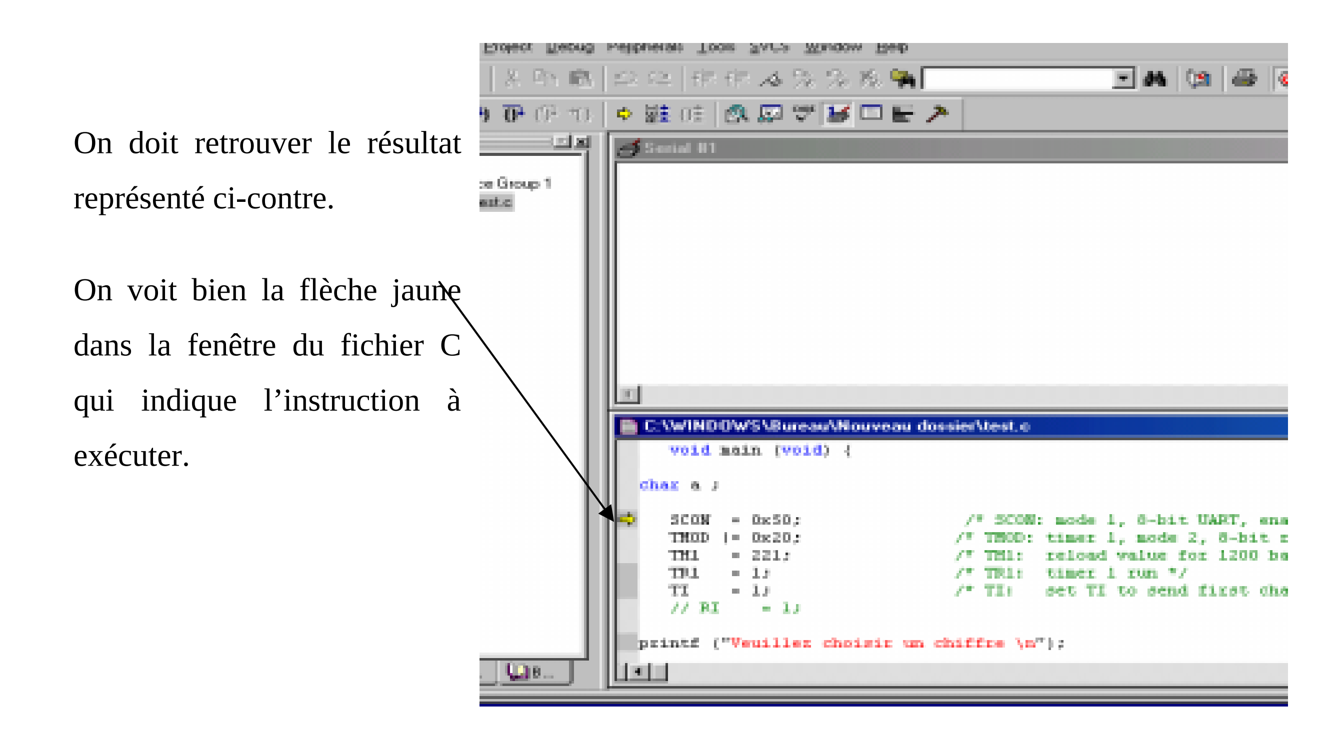


Dans ce mode, il est possible d'exécuter le programme pas à pas, de simuler un grand nombre de périphériques et visualiser le contenu des registres et des mémoires après chaque instruction. Dans notre exemple, on s'intéressera au port série. Keil dispose d'une interface graphique (fenêtre) : Toute donnée envoyée au port série est affichée sur cette interface et toute donnée saisie au niveau de l'interface est considérée comme étant reçue sur le port série.

Il faut activer la fenêtre simulant le port série avec le menu ***View*** >> ***Serial Window #1*** et ensuite l'afficher en même temps que le fichier C avec le menu ***Window*** >> ***Tile Horizontally***.

On doit retrouver le résultat représenté ci-contre.

On voit bien la flèche jaune dans la fenêtre du fichier C qui indique l'instruction à exécuter.



La Séléction de la fenêtre du fichier C permet de la rendre active. Nous pouvons alors commencer à exécuter le programme pas à pas avec le menu ***Debug >> Step*** ou la touche ***F11***. On constatera qu'après chaque instruction la flèche se déplace vers l'instruction suivante ; en arrivant à l'instruction *printf ( ),* le message apparaîtra dans la fenêtre du port série. Au niveau de l'instruction *scanf ( )* l'exécution sera bloquée, en effet la fonction *scanf* met le programme (le microcontrôleur) en état d'attente et il ne passe à l'instruction suivante qu'après la réception d'une donnée sur le port série. Pour simuler la réception d'une donnée, il suffit d'activer la fenêtre du port série en cliquant dedans avec la souris et en tapant une touche du clavier. L'exécution pourra alors continuer et on verra s'afficher sur la fenêtre du port série le message correspondant au résultat du traitement de la donnée saisie au clavier.

IV.2 Adaptation du programme à la configuration matérielle.

Le faite que la simulation s'est déroulée sans erreurs permet de conclure avec une grande probabilité que le programme est correcte mais ne signifie pas absolument qu'il s'exécutera exactement de la même manière dans la carte cible. La différence d'exécution du programme dans les deux phases (simulation et après chargement)

est due principalement au faite que le compilateur ne dispose pas suffisamment d'informations concernant :

- La configuration matérielle de la carte (espaces d'adressage, tailles des mémoires, etc..).
- Les valeurs d'initialisation des registres au lancement du programme.
- Les ressources utilisées par le moniteur comme dans notre cas.

Ainsi, le compilateur utilise des paramètres par défaut qui différent généralement de ceux de la plate forme matérielle utilisée. Ce qui conduit logiquement à un comportement imprévisible du programme une fois chargé sur la carte. Dans notre cas par exemple :

- Le moniteur occupe la zone du CODE 0000…7FFF h, ce qui signifie qu'on doit indiquer au compilateur que cette zone ne peut être utilisée pour loger le programme généré.
- La mémoire U3 occupe la plage d'adresses 8000…FFFF h des zones CODE et DATA en même temps. L'utilisation de plages communes pour le code et les données provoque l'écrasement de l'un par l'autre.

Pour remédier à ce problème, Keil fournit un fichier ***startup.a51*** se trouvant dans le répertoire C51\Lib et qui contient du code prêt à être paramètré par un utilisateur pour l'adapter à la configuration matérielle.

Ce fichier contient des sections pour :

- Déterminer la taille totale des mémoires internes et externes.
- Déterminer les valeurs d'initialisation de ces mémoires.
- Déterminer l'adresses à partir de la quelle le code doit être logé.
- Déterminer les valeurs d'initialisation des registres 'importants' (tel que **SP** :Stack Pointer).
- Etc.…

Pour l'utiliser, il faut en faire une copie dans le répertoire de travail et l'ajouter au projet.

Ainsi, pendant la simulation, on dispose de conditions semblables à celles du matériel et les résultats obtenus durant cette phase peuvent être considérés comme

concluants. Il faut noter que certains de ces paramètres peuvent être définis directement à travers l'interface graphique de Keil et c'est ce qu'on va faire.

Commencer par sélectionner ***Options for Target 'Target 1'*** à partir du menu déroulant de Target1 dans la fenêtre gauche (voir figure)



Dans la fenêtre qui apparaîtra, sélectionner l'onglet **Target** et choisir les paramètres suivants :

Xtal (Mhz) :11.059 qui est la fréquence réelle utilisée dans la carte.

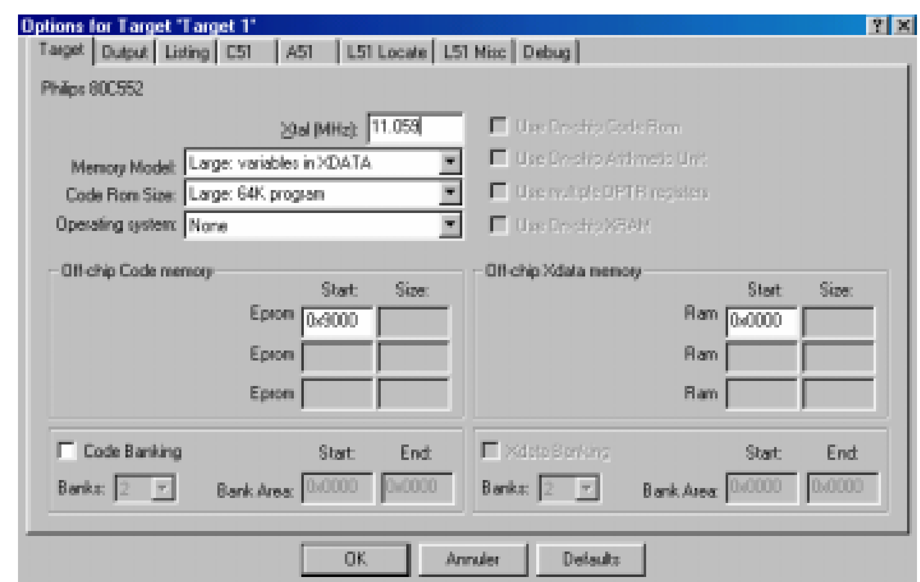
Memory Model : Large variables in XDATA.

Code Rom Size : Large 64K program.

Operating system : None

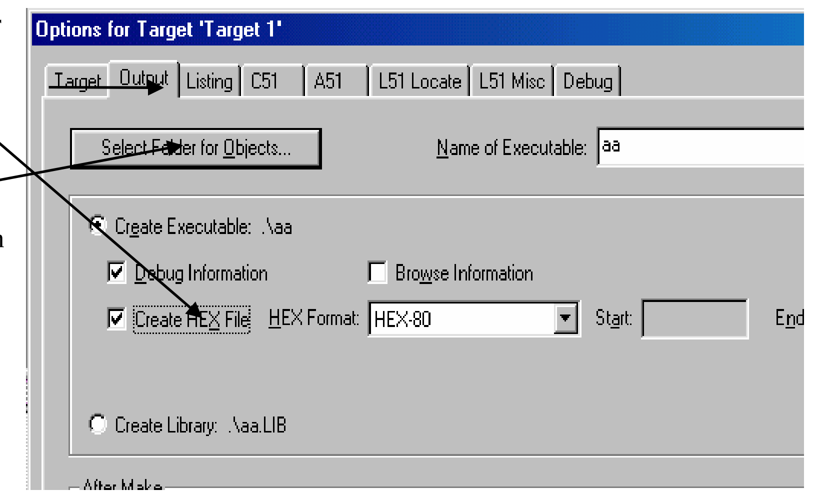
Eprom : 0x9000. En effet le moniteur utilisant les 32 Ko inférieurs, on ne peut utiliser que la zone d'adressage à partir de 8000h ; on a choisi 9000h.

Ram : 0x0000



Sélectionner l'onglet **Output** et cocher la case *Create HEX File* pour que le compilateur génère un fichier au format Hexadécimal pouvant être chargé dans le carte.

Il est possible de changer la destination d'enregistrement du fichier .HEX avec le bouton *Select folder for Objects…*

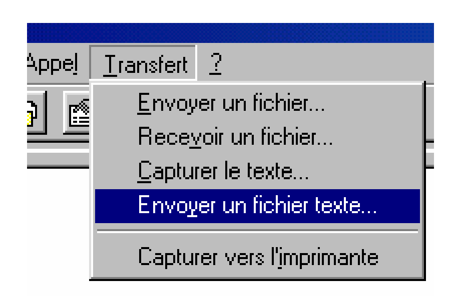


Après avoir recompilé le programme on devrait avoir un fichier .HEX prêt à être chargé dans la zone code à partir de l'adresse 9000h.

IV.3 Chargement du programme.

Après avoir lancé le logiciel de transfert et relié le PC à la carte et avoir mis la carte sous tension, le chargement se fait en quatre étapes :

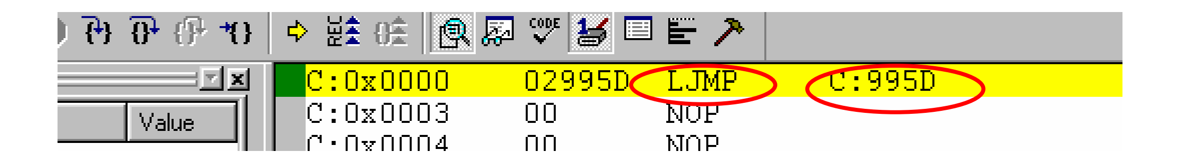
1. On indique au moniteur qu'on va lui transmettre un fichier .HEX avec la commande **H** : taper la lettre h dans l'interface d'hyperterminal.
2. Le microcontrôleur (moniteur) renvoie la lettre H pour indiquer qu'il est prêt.
3. On envoie le fichier .HEX à charger en utilisant la commande ***Transfert>>Envoyer un fichier texte..*** du menu de *HYPERTERMINAL* comme indiqué ci-dessous.



4. Le microcontroleur (moniteur) indique le nombre d'octets reçus si le transfert s'est déroulé sans incidents ou affiche un message d'erreur dans le cas contraire.

IV.4 Exécution du programme.

Une fois le programme chargé, on peut l'exécuter en utilisant la commande '**G**' suivie de l'adresse de la première instruction à exécuter. Si par exemple la première instruction du programme se trouve à l'adresse 8000h ; on l'exécute avec la commande **G 8000**. Dans notre exemple nous avons chargé le programme à partir de l'adresse **9000h**. Mais ça ne veut pas dire que la première instruction se trouve à cette adresse, en effet le compilateur commence par placer le code correspondant aux fonctions utilisées *(printf et scanf dans notre cas)* et puis le code du programme proprement dit. Ce problème peut être résolu grâce à une option qui permet d'avoir les instructions assembleur générées ainsi que leur emplacement mémoire dans la fenêtre '**Disassembly Window**'. Et on trouve à l'adresse **0x0000** une instruction de saut vers la première instruction du programme. Pour avoir cette adresse, passer en mode débogage et sélectionner le menu ***View*** >> ***Disassembly Window*** . Déplacer le curseur dans la fenêtre qui apparaîtra jusqu'à l'adresse C :0x0000. On trouvera l'instruction de saut (LJMP : Long Jump) ainsi que l'adresse de la première instruction. Dans notre exemple l'adresse de début est : **995D**.



L’exécution du programme se fait alors avec la commande **G 995D**.

I. Introduction :

Le 8051 est un microcontrôleur qui a été développé par Intel et autour duquel, par la suite, ont été conçus d'autres microcontrôleurs qui sont alors dit de la famille 8051. Plusieurs constructeurs font aujourd'hui des microcontrôleurs de cette famille (se reporter au chapitre sur les microcontrôleurs de la Famille 8051). Je vais particulièrement m'intéresser ici au noyau 8051 et 8052 ainsi qu'aux 80C535 et 80C537 de Siemens, microcontrôleurs très puissants. Il est alors important de s'intéresser à l'architecture du noyau 8051 qui est donc commune à tous les microcontrôleurs de la famille, au détail près que sur des versions plus perfectionnées, il y a ajout de fonctions. Notons au passage que le cycle machine dure 12 fois la période de l'oscillateur utilisé. Voyons aussi une présentation matérielle de ces microcontrôleurs, accompagnée d'un exemple de mise en oeuvre avec de mémoire de programme et de données externes. J'ai choisi de présenter le 8031, 8032 ainsi que le SAB80C535 et 80C537 de Siemens car ce sont les micros les plus courants dans cette famille.
Il est tant maintenant de présenter le jeu d'instructions du 8051. J'ai mis à disposition sur ce site un assembleur ainsi qu'un simulateur. L'assembleur est un freeware et il est à mon avis le meilleur assembleur shareware car il respecte à 100% l'assembleur Intel. De plus il est compatible avec plus de 30 microcontrôleurs de la famille.... Le mieux pour juger, c'est de le charger: asem51.zip
Quant au simulateur, il s'agit d'un simulateur sous DOS en mode texte. Ce n'est qu'une version d'évaluation d'un soft payant mais il n'y a pas de restriction d'utilisation : emily.zip
Intéressons nous maintenant à la structure de la RAM interne du 8051, puis à l'utilisation des timers, du port série, des interruptions; du convertisseur A/N et de l'unité arithmétique du 80C537.

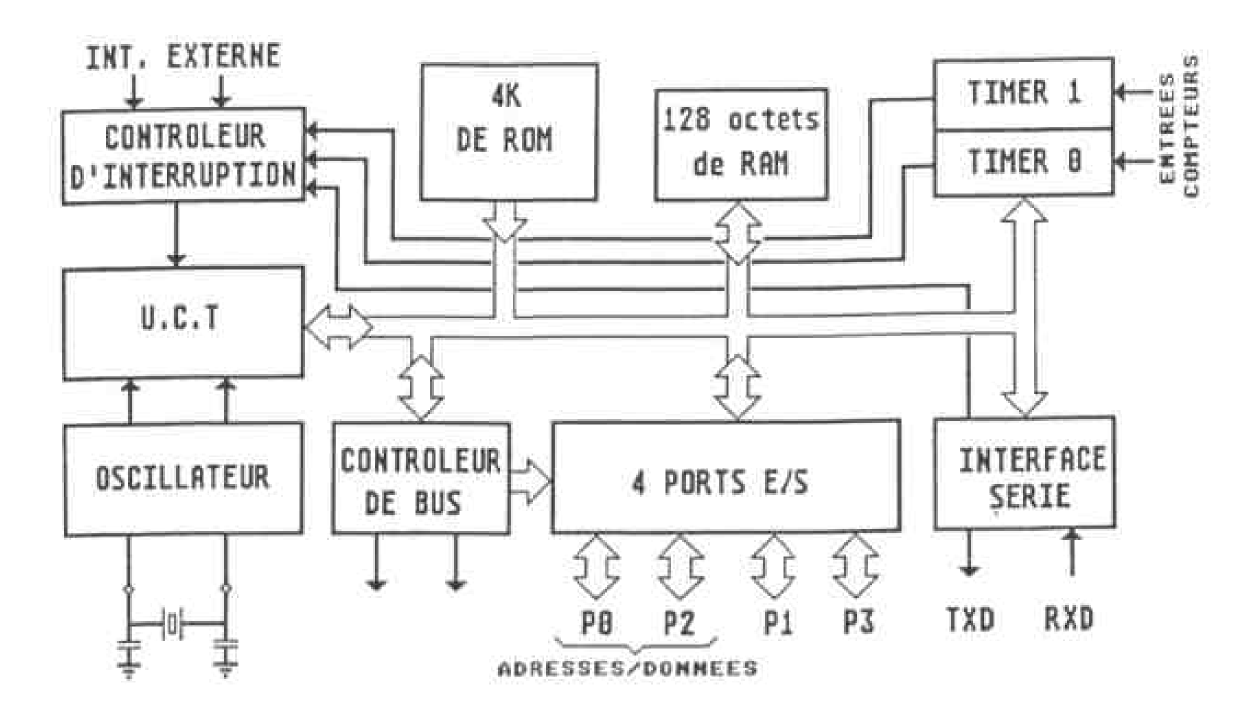
Nom	Marque	Boîtier	RAM	E/S Parallèle	E/S Série	A/D (bits)	Timers	Source Int
80C31	*	DIP40	128	32	1 port		2X16bits	5
80C32	*	DIP40	256	32	1 port		3X16bits	6
80C198	Intel	PLCC52	232	16E/S + 4E		10	1X16 + WD	28
80C196	Intel	PLCC68	232	16E/S + 8E	1 port	10	1X16 + WD	28
80C152	Intel	NC	256	40E/S	2 ports		2X16 bits	11

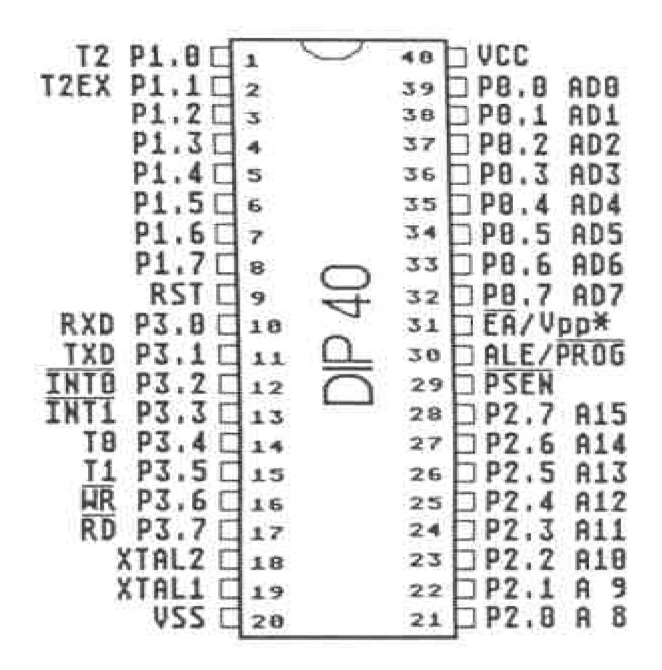
80C451	Intel	NC	128	56E/S	1 port		2X16 bits	5
80C535	Siemens	PLCC68	256			8	3X16 + WD	
80C537	Siemens	PLCC84	256			8	3X16 + WD	
80C557	Philips	PLCC						

L'environnement de base du noyau 8051 est constitué de :

- 32 lignes d'E/S bidirectionnelles réparties en 4 ports : P0, P1, P2, P3
- 128 octets de RAM interne
- 2 TIMERS 16 bits, T0 et T1 fonctionnant suivant 4 modes
- une interface de communication série UART
- une unité de contrôle gérant 5 interruptions selon 2 niveaux de priorité
- un circuit d'horloge embarqué nécessitant un quartz externe

On peut schématiser ce noyau de la façon suivante :





II. Le 8031 et le 8032

Ces deux composants ont le même brochage. On remarque que de nombreuses pattes ont des fonctions secondaires. Etudions ce brochage de plus près:

- Entrée /EA : (**E**xternal **A**ccess) si EA=0, les instructions sont recherchées dans la mémoire programme externe.
- RST : Entrée d'initialisation. Un état haut pendant deux cycles machines sur cette broche entraîne une initialision du microcontrôleur.
- Sortie /PSEN : (**P**rogramm **S**tore **EN**able) passe à 0 lorsque le micro va rechercher une instruction en mémoire programme externe.
- Sortie ALE : (**A**dress **L**atch **E**nable) prévue pour commander le démultiplexage du port P0. Si ALE est à 1, P0 présente la partie A0 à A7 du bus d'adresse et si ALE est à 0, P0 sert de bus de donnée. Pour mieux comprendre, se reporter à l'organisation du bus.
- XTAL1 et XTAL2 : Placer le quartz entre ces deux broches avec deux condensateurs de 22pF entre ces deux broches et la masse.
- P0.0 à P0.7 : 8 lignes du port P0 du type "à drain ouvert". Si ces lignes sont utilisées en sortie, il est nécessaire de le doter de résistances de rappel. Se reporter au paragraphe l'organisation du bus.
- P1.0 à P1.7 : Port bidirectionnel avec résistances de rappel au +5V intégrées.
- P2.0 à P2.7 : Idem que port P1 sauf : fonction secondaire du port: adresses de A8 à A15. Se reporter au paragraphe l'organisation du bus.
- P3.0 à P3.7 : Idem que port P1 sauf : fonctions secondaires :
- P3.0 : RxD entrée de l'interface série
- P3.1 : TxD sortie de l'interface série
- P3.2 : /INT0 entrée pour interruption externe
- P3.3 : /INT1 Idem
- P3.4 : T0 entrée de comptage pour timer0
- P3.5 : T1 entrée de comptage pour timer1
- P3.6 : /WR sortie écriture de la mémoire externe
- P3.7 : /RD sortie lecture de la mémoire externe

III. Lejeu d'instructions du 8051.

Signification des symboles utilisés:

Rn	Un des registres actifs R0 à R7
direct	Adresse d'un octet de RAM interne, d'un port, ou SFR
@Ri	Adressage indirect par registre R0 ou R1
#data	Donnée 8 bits
#data16	Donnée 16 bits
bit	Adresse au niveau du bit
rel	Adresse relative au PC en complément à 2 de -128 à +127
addr11	Adresse limitée au bloc de 2Ko dans lequel figure l'instruction
addr16	Adresse sur l'espace de 64Ko

- L'instruction MOV : (MOV <dest>, <source>)

Mnémonique	Syntaxe	Code	octets	cycle
MOV	A, Rn	E8+n	1	1
MOV	A, direct	E5	2	1
MOV	A, @Ri	E6+i	1	1
MOV	A, #data	74	2	1
MOV	Rn, A	F8+n	1	1
MOV	Rn, direct	A8+n	2	2
MOV	Rn, #data	78+n	2	1
MOV	direct, A	F5	2	1
MOV	direct, Rn	88+n	2	2

MOV	direct, direct	85	3	2
MOV	direct, @Ri	86+i	2	2
MOV	direct, #data	75	3	2
MOV	@Ri, A	F2+i	1	1
MOV	@Ri, direct	A6+i	2	2
MOV	@Ri, #data	76+i	2	1
MOV	DPTR, #data16	90	3	2

- L'instruction MOVC : (MOVC A, @A+<base-reg>) permet de lire un octet dans la mémoire pgm.

MOVC	A,@A+DPTR	93	1	2
MOVC	A,@A+PC	83	1	2

- L'instruction MOVX : (MOVX <dest>, <source>) permet la lecture ou l'écriture d'un octet en RAM externe.

MOVX	A,@Ri	E2+i	1	2
MOVX	A,@DPTR	E0	1	2
MOVX	@Ri,A	F2+i	1	2
MOVX	@DPTR,A	F0	1	2

- Les instructions PUSH et POP permettent respectivement de sauvegarder sur la pile ou d'y récupérer des données.

PUSH	direct	C0	2	2
POP	direct	D0	2	2

- L'instruction XCH : (XCH A, <byte>) échange les données de l'accumulateur A et de l'octet adressé.

XCH	A, Rn	C8+n	1	1
XCH	A, direct	C5	2	1
XCH	A, @Ri	C6+i	1	1

- L'instruction XCHD : (XCHD A, @Ri) échange les quartets de poids faible entre l'accu A et l'octet adressé.



XCHD	A, @Ri	D6+i	1	1

- L'instruction ADD : (ADD A, <byte>) additionne un octet et l'accu A, résultat dans A.

ADD	A,Rn	28+n	1	1
ADD	A, direct	25	2	1
ADD	A, @Ri	26+i	1	1
ADD	A, #data	24	2	1

- L'instruction ADDC : (ADDC A, <byte>) additionne un octet, l'accu A et la retenue, résultat dans A.

ADDC	A, Rn	38+n	1	1
ADDC	A, direct	35	2	1
ADDC	A, @Ri	36+i	1	1
ADDC	A, #data	34	2	1

- L'instruction SUBB : (SUBB A, <byte>) soustrait un octet ainsi que la retenue au contenu de A, résultat dans A.

SUBB	A, Rn	98+n	1	1
SUBB	A, direct	95	2	1
SUBB	A, @Ri	96+i	1	1
SUBB	A, #data	94	2	1

- L'instruction INC : (INC <byte>) incrémente un octet ou DPTR.

INC	@Ri	06+i	1	1
INC	A	04	1	1
INC	direct	05	2	1
INC	Rn	08+n	1	1
INC	DPTR	A3	1	2

- L'instruction DEC : (DEC <byte>) décrémente un octet.

DEC	@Ri	16+i	1	1
DEC	A	14	1	1
DEC	direct	15	2	1
DEC	Rn	18+n	1	1

- L'instruction MUL : (MUL AB) multiplie l'accu A et le registre B, résultat : octet faible dans A et octet fort dans B.
- L'instruction DIV : (DIV AB) divise le contenu de A par le contenu de B, quotient dans A et reste dans B.
- L'instruction DA : (DAA) ajustement décimal de A.

MUL	AB	A4	1	4
DIV	AB	84	1	4



DA	A	D4	1	1

- L'instruction ANL : (ANL <dest>, <source>) réalise un ET logique entre source et dest, résultat dans dest.

ANL	A, #data	54	2	1
ANL	A, @Ri	56+i	1	1
ANL	A, direct	55	2	1
ANL	A, Rn	58+n	1	1
ANL	direct, #data	53	3	2
ANL	direct, A	52	2	1
ANL	C, /bit	B0	2	2
ANL	C, bit	82	2	2

- L'instruction ORL : (ORL <dest>, <source>) réalise un OU logique entre source et dest, résultat dans dest.

ORL	A, #data	44	2	1
ORL	A,@Ri	46+i	1	1
ORL	A, direct	45	2	1
ORL	A, Rn	48+n	1	1
ORL	direct, #data	43	3	2
ORL	direct, A	42	2	1
ORL	C, /bit	A0	2	2
ORL	C, bit	72	2	2

- L'instruction XRL : (XRL <dest>, <source>) réalise un OU exlusif logique entre source et dest, résultat dans dest.

XRL	A, #data	64	2	1
XRL	A, @Ri	66+i	1	1
XRL	A, direct	65	2	1
XRL	A, Rn	68+n	1	1
XRL	direct, #data	63	3	2
XRL	direct, A	62	2	1

- L'instruction CLR : (CLR <A/bit>) met A ou un bit à 0

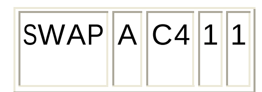
CLR	A	E4	1	1
CLR	bit	C2	2	1
CLR	C	C3	1	1

- L'instruction CPL : (CPL <A/bit>) complémente A ou un bit

CPL	A	F4	1	1
CPL	bit	B2	2	1
CPL	C	B3	1	1

- L'instruction RL : (RL A) rotation vers la gauche du contenu de A
- L'instruction RLC : (RLC A) rotation vers la gauche du contenu de A+retenue
- L'instruction RR : (RR A) rotaion vers la droite du contenu de A
- L'instruction RRC : (RRC A) rotation vers la droite du contenu de A+retenue
- L'instruction SWAP : (SWAP A) échange le quartet de poids faible avec celui de poids fort de A

RL	A	23	1	1
RLC	A	33	1	1
RR	A	03	1	1
RRC	A	13	1	1



SWAP	A	C4	1	1

- L'instruction SETB : (SETB <bit>) met à 1 un bit

SETB	bit	D2	2	1
SETB	C	D3	1	1

- L'instrucion MOV : (MOV <bitdest>, <bitsrc>) copie le bitsrc dans le bitdest

MOV	bit, C	92	2	2
MOV	C, bit	A2	2	1

- L'instruction ACALL : réalise un saut absolu inconditionnel
- L'instruction LJMP : réalise un saut long inconditionnel
- L'instruction SJMP : réalise un saut court par adressage relatif
- L'instruction JMP : réalise un saut indirect

AJMP	addr11	E1	2	2
LJMP	addr16	02	3	2
SJMP	rel	80	2	2
JMP	@A+DPTR	73	1	2

- L'instruction JZ : saut si A=0
- L'instruction JNZ : saut si A<>0
- L'instruction JC : saut si retenue à 1
- L'instruction JNC : saut si retenue à 0
- L'instruction JB : saut si bit à 1
- L'instruction JNB : saut si bit à 0
- L'instruction JBC : saut si le bit est à 1 et mise à zero de celui-ci

JZ	rel	60	2	2
JNZ	rel	70	2	2
JC	rel	40	2	2

JNC	rel	50	2	2
JB	bit, rel	20	3	2
JNB	bit, rel	30	3	2
JBC	bit,rel	10	3	2

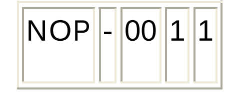
- L'instruction CJNE : (CJNE <byte1>, <byte2>, <rel>) saut si byte1 et byte2 sont différents

CJNE	@Ri, #data, rel	B6+i	3	2
CJNE	A, #data, rel	B4	3	2
CJNE	A, direct, rel	B5	3	2
CJNE	Rn, #data, rel	E8+n	3	2

- L'instruction DJNZ : (DJNZ <byte>, rel) décrémente byte et saut si résultat différent de 0

DJNZ	direct, rel	D5	3	2
DJNZ	Rn, rel	D8+n	2	2

- L'instruction NOP : pas d'opération



NOP	-	00	1	1

IV. Organisation de la mémoire interne.

La mémoire occupe l'espace d'adresses 00 à 7FH (ou 00 à 0FFH pour 8052). Les registres à fonction spéciale (SFR) occupent dans tous les cas l'espace de 7FH à 0FFH. Dans le cas où il y a 256 octets de RAM (8052 par exemple), une partie de la RAM se superpose avec les SFR. Mais les SFR sont alors disponibles qu'en adressage direct et la RAM n'est accessible qu'en adressage indirect (par pointeur)

IV.1 Les registres universels :

Les 32 premiers octets de RAM interne peuvent être utilisés en tant que registres universels. Mais seuls 8 de ces 32 octets peuvent être actifs. On parle alors de 4 banques de 8 registres, l'activation de la banque se faisant à l'aide des deux bits RS0 et RS1 du registre PSW.

adresse	nom	Banque	RS1	RS0
1FH 18H	R7 R0	3	1	1
17H 10H	R7 R0	2	1	0
0FH 08H	R7 R0	1	0	1
07H 00H	R7 R0	0	0	0

IV.2 Les Zones mémoire adressable au niveau du bit :

16 octets peuvent être adressés au niveau du bit (de 20H à 2FH). L'adresse de chacun des bits est exprimée à l'aide d'un octet :

Adresse	b7	b6	b5	b4	b3	b2	b1	b0
2FH	7F	7E	7D	7C	7B	7A	79	78
2EH	77	76	75	74	73	72	71	70
2DH	6F	6E	6D	6C	6B	6A	69	68
2CH	67	66	65	64	63	62	61	60
2BH	5F	5E	5D	5C	5B	5A	59	58
2AH	57	56	55	54	53	52	51	50
29H	4F	4E	4D	4C	4B	4A	49	48
28H	47	46	45	44	43	42	41	40

27H	3F	3E	3D	3C	3B	3A	39	38
26H	37	36	35	34	33	32	31	30
25H	2F	2E	2D	2C	2B	2A	29	28
24H	27	26	25	24	23	22	21	20
23H	1F	1E	1D	1C	1B	1A	19	18
22H	17	16	15	14	13	12	11	10
21H	0F	0E	0D	0C	0B	0A	09	08
20H	07	06	05	04	03	02	01	00

IV.3 Cartographie des SFR d'un 8031 (et 8032 indiqué par un *)

Symbole	Fonction	Adresse	Adresse au niveau du bit								Etat initial
B	reg. pour mul. et div.	F0H	F7	F6	F5	F4	F3	F2	F1	F0	00000000
ACC	Accumulateur	E0H	E7	E6	E5	E4	E3	E2	E1	E0	00000000
PSW	registre d'état	D0H	D7	D6	D5	D4	D3	D2	D1	D0	00000000
			CY	AC	F0	RS1	RS0	0V	---	P	
TH2*	Poids fort du Timer 2	CEH									00000000
TL2*	poids faible du Timer 2	CDH									00000000
RCAP2H*	Capture/rech du T2	CCH									00000000
RCAP2L*	Idem	CBH									00000000

T2CON*	Contrôle du T2	C8H	CF	CE	CD	CC	CB	CA	C9	C8	00000000
			TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2	CP/RL2	
IP	priorité des interruptions	B8H	BF	BE	BD	BC	BB	BA	B9	B8	xx000000
			---	---	PT2	PS	PT1	PX1	PT0	PX0	
P3	Port P3	B0H	B7	B6	B5	B4	B3	B2	B1	B0	11111111
			RD	WR	T1	T0	INT1	INT0	TxD	RxD	
IE	Validation des int.	A8H	AF	AE	AD	AC	AB	AA	A9	A8	0x000000
			EA	---	ET2	ES	ET1	EX1	ET0	EX0	
P2	Port P2	A0H	A7	A6	A5	A4	A3	A2	A1	A0	11111111
			A15	A14	A13	A12	A11	A10	A9	A8	
SBUF	Données du port série	99H									00000000
SCON	Contrôle du port série	98H	9F	9E	9D	9C	9B	9A	99	98	00000000
			SM0	SM1	SM2	REN	TB8	RB8	TI	RI	
P1	Port P1	90H	97	96	95	94	93	92	91	90	11111111
			---	---	---	---	---	---	T2EX	T2	
TH1	Poids fort du Timer 1	8DH									00000000
TH0	Poids fort du Timer 0	8CH									00000000
TL1	poids faible du T1	8BH									00000000
TL0	poids faible du T0	8AH									00000000

TMOD	Modes pour T0 et T1	89H	-------- GATE	----- C/T	---- M1	---- M0	-------- GATE	----- C/T	---- M1	---- M0	00000000
TCON	Contrôle de T1 et T2	88H	8F TF1	8E TR1	8D TF0	8C TR0	8B IE1	8A IT1	89 IE0	88 IT0	00000000
PCON	Mode de consommation	87H	-------- SMOD	-- --	-- --	-- --	----- GF1	----- GF0	---- PD	----- IDL	0xxxxxxx
DPH	Poids fort de DPTR	83H									00000000
DPL	poids faible de DPTR	82H									00000000
SP	Pointeur de pile	81H									00000111
P0	Port P0	80H	87 AD7	86 AD6	85 AD5	84 AD4	83 AD3	82 AD2	81 AD1	80 AD0	11111111

IV.4 SFR supplémentaires (ou changées) des 80C535 par rapport aux 8032 :

Symbole	Fonction	Adresse	Adresse au niveau du bit								Etat Initial
P5	Port P5	F8H	FF	FE	FD	FC	FB	FA	F9	F8	11111111
P4	Port P4	E8H	EF	EE	ED	EC	EB	EA	E9	E8	11111111
P6	Port P6 (8E)	DBH									-----------
DAPR	Contrôle du CAN	DAH	VinAREF				VinAGND				00000000

ADDAT	Donnée du CAN	D9H		00000000
ADCON	Contrôle de CAN	D8H	DF DE DD DC DB DA D9 D8 BD CLK (vide) BSY ADM MX2 MX1 MX0	00000000
TH2	Poids fort de T2	CDH		00000000
TL2	poids faible de T22	CCH		00000000
CRCH	registre 16 bits de	CBH		00000000
CRCL	capture, cmp et rech.	CAH		00000000
T2CON	Contrôle du T2	C8H	CF CE CD CC CB CA C9 C8 T2PS I3FR I2FR T2R1 T2R0 T2CM T2I1 T2I0	00000000
CCH3	Registre de capture	C7H		00000000
CCL3	et de comp. 3	C6H		00000000
CCH2	Registre de capture	C5H		00000000
CCL2	et de comp 2	C4H		00000000
CCH1	Registre de capture	C3H		00000000
CCL1	et de comp 1	C2H		00000000
CCEN	Validation capt/Cmp	C1H	COCAH3 COCAL3 -2 -2 -1 -1 COCAH0 COCAL0	00000000
IRCON	Contrôle des Int	C0H	C7 C6 C5 C4 C3 C2 C1 C0	00000000

			EXF2	TF2	IEX6	IEX5	IEX4	IEX3	IEX2	IADC	
IP1	Priorité des Int	B9H									00000000
					IP1.5	IP1.4	IP1.3	IP1.2	IP1.1	IP1.0	
IEN1	Validation de Int	B8H	BF	BE	BD	BC	BB	BA	B9	B8	00000000
			EXEN2	SWDT	EX6	EX5	EX4	EX3	EX2	EADC	
IP0	Priorité des Int	A9H									00000000
					IP0.5	IP0.4	IP0.3	IP0.2	IP0.1	IP0.0	
IEN0	Validation des Int	A8H	AF	AE	AD	AC	AB	AA	A9	A8	00000000
			EAL	WDT	ET2	ES	ET1	EX1	ET0	EX0	
PCON	Mode de consommation	87H									0xxxxxxx
			SMOD	PDS	IDLS		GF1	GF0	PDE	IDLE	

Bibliographie

Microcontrôleurs 8051 et 8052 : Description et mise en œuvre

Edition Dunod : juillet 2001

Auteur : Bernard Odant

Microcontroller Projects in C for the 8051

Edition: Newnes: 2000

Auteur: Dogan Ibrahim

Microcontrôleurs : principes et applications

Edition Radio : Paris 1991

Auteur : Christian Tavernier

The indispensable PC Hardware book

Edition Addison-Wesley 2001

Auteur: Hans Peter Messmer

Mise en Oeuvre et applications du microcontrôleur 8051

Edition Masson , Paris 1996

Auteur : P. KAUFFMANN

Le microprocesseur Pentium : Architecture et programmation

Edition : Armand Colin 1998

Auteur : Patrick Cohen

Microprocesseurs du CISC au RISC,

Edition Dunod 1998

auteur: Henri LILEN

80386 architecture et circuit associés

Edition Eyrolles 1998

Auteurs: Eric Boit, Fabienne Niess-Gerber, Roland Dubois

Turbo Assembler user's guide

Borland International 1995

Programmer en assembleur sur PC

Editions Mico application 1995

Auteur: Holger SCHAKEL

Ingénierie des systèmes à microprocesseur

Edition Masson 1995

Auteurs: Eric Martin, Jean luc Philippe

Microsoft Macro Assembler Reference

Version 6.0 for MS OS/2 and MSDOS

Edition 1995

Borland C++ 5.0

Guide du programmeur, Structure du langage, Techniques de programmation avancées

Implémentation du C ANSI

Edition 1996.

http://www.jls-info.com/julien/8051/

asem51.zip, broch32.jpg, brochage.htm, arith.htm, interruption.htm, serie.htm, timer.htm

http://perso.wanadoo.fr/michel.hubin/physique/microp/chap_mp1.htm#debut#debut

microprocesseur, microcontrôleur, composant ASIC ou FPGA, informatique industrielle

http://perso.wanadoo.fr/e-lektronik/LEKTRONIK/C9A.htm#C9A2

Qu'est-ce qu'un microcontrôleur?, Comment fonctionne un microcontrôleur?, La programmation d'un microcontrôleur, A quoi servent les microcontrôleurs?

http://www.ybet.be/hardware/hardware1.htm

Structure interne d'un processeur de base, Mémoire,Passage du CISC au RISC,

Architecture pipeline, superpipeline et superscalaire,Architecture interne des processeurs, Architecture des Athlon 64 et Athlon XP

Décodage d'adresses, Types de mémoires

http://www.jbaumann.ch

Historique des microprocesseurs,Gestion de la mémoire, Les ports d'entrée / sortie (E/S)

Interruptions, E/S pilotées par interruption, Accès direct à la mémoire (DMA)

http://www.polytech-lille.fr/~rlitwak/Cours_MuP

Microprocesseurs et micro-ordinateurs:Définitions, Architecture et constituants , Application des microprocesseurs.

Composants d'un microprocesseur:L'unité arithmétique et logique (UAL)

http://www-gtr.iutv.univ-paris13.fr/Cours

Introduction au langage machine, Branchements, Déclaration d'un segment en assembleur, Notion de pile, Les fonctions du BIOS, Vecteurs d'interruptions, Les bus du PC

http://www.qsl.net/f6flv/cntr.html

Le 8259 maitre (IRQ0 a IRQ7), Le 8259 esclave (IRQ8 a IRQ15), Affectation des interruptions. Registres et programmation du 8259

http://www.qsl.net/f6flv/docbios.html

Donnees Timer, Donnees Video, Donnees unite de disquette, Donnees clavier, Interfaces V24/RS232C

Tableau des primitives, Primitives du BIOS, Messages du Bios, Les Secrets du BIOS PC/AT

http://inferno.cs.univ-paris8.fr/~am/tutorial/os/tutorial07.html

Le processeur et les interruptions. Des interruptions matérielles gérées par un chipset : le 8259A (1)

Des interruptions matérielles gérées par un chipset : le 8259A (2)

Paramétrer le contrôleur d'interruptions:

ICW1 (port 0x20 and 0xA0),ICW2 (port 0x21 et 0xA1), ICW3 (port 0x21 et 0xA1)

ICW4 (port 0x21 et 0xA1), OCW1 (ports 0x21 et 0xA1), OCW2 (ports 0x20 et 0xA0)

OCW3 (ports 0x20 et 0xA0)

http://www.uv.tietgen.dk/staff/mlha/pc/prog/asm/int/21

MASM:INT21, INT21.DOC,..,DEBUG,MASM,WASM,asm.doc

Warm Boot (~ Ctrl-Alt-Del),BOOT.COM, WARM.COM, Cold Boot (~ Reset)

http://www.nvg.ntnu.no/bbc/doc/Master%20512%20Technical%20Guide/m512techb_int21.htm

Function 0- Program terminate, Function 1- Character input with echo,...,

Function 50h - Get disc information (Undocumented call),Fundion 56h - Rename file,Function 57h - Get or set file date and time

http://physics.nist.gov./cuu/Units/binary.html